**xBRC Internal Design Notes**

**Revision History**

|  |  |  |  |
| --- | --- | --- | --- |
| **Rev** | **Date** | **Author** | **Description** |
| 1.0 | 5/1/12 | Manny Vellon | Release Version |
| 1.1 | 5/22/2012 | Arkady Glabek | Added Vehicle Association |
| 1.2 | 8/15/2012 | Arkady Glabek | Added /etc/nge/config/environment.properties |
| 1.3 | 4/12/2013 | Arkady Glabek | Added config properties: updatestreamfasttaps, fasttapurl, readerupdatestreams. Removed obsolete park entry sections. |
| 1.4 | 7/30/2013 | Arkady Glabek | Added the Scheduler section. |

**Document Approvers & Sign-Off**

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Approver** | **Role** | **Document Accept/Reject** |
|  |  |  |  |

Table of Contents

[1 Introduction 4](#_Toc353535964)

[1.1 Purpose 4](#_Toc353535965)

[1.2 Scope 4](#_Toc353535966)

[1.3 Assumptions and Constraints 4](#_Toc353535967)

[1.4 Reference 4](#_Toc353535968)

[2 Internal Design Notes 4](#_Toc353535969)

[2.1 Code Structure 4](#_Toc353535970)

[2.2 Major Components 5](#_Toc353535971)

[2.3 xBRC Major Components 6](#_Toc353535972)

[2.4 Establishing Reader-to-xBRC Communication - the Reader Management Component 7](#_Toc353535973)

[2.5 The Event Pipeline 8](#_Toc353535974)

[2.6 Receiving raw Reader TAP or GPIO events 13](#_Toc353535975)

[2.7 Housekeeping Components 13](#_Toc353535976)

[3 Model-Specific Information 18](#_Toc353535977)

[3.1 Attraction Model 19](#_Toc353535978)

[3.2 Space Model 21](#_Toc353535979)

[3.3 Park Entry Model 22](#_Toc353535980)

[4 Web User Interface 22](#_Toc353535981)

[5 xBRC Configuration Values 24](#_Toc353535982)

[5.1 *ControllerInfo* Settings 24](#_Toc353535983)

[5.2 *ESBInfo* Settings 25](#_Toc353535984)

[5.3 Model-SpecificSettings 25](#_Toc353535985)

[6 Installing the xBRC Code 27](#_Toc353535986)

[6.1 Installing and Updating xBRC Code 27](#_Toc353535987)

[6.2 Location of xBRC Files and Directories 27](#_Toc353535988)

[6.3 Creating the xBRC Database and Configuring /etc/nge/config/environment.properties 29](#_Toc353535989)

# Introduction

## Purpose

This document describes the internal structure of the xBRC. The goal of the document is to identify the xBRCs main areas of functionality and to provide enough context and structure to drive its testing effort. Additionally, this document will server to educate new developers learning the xBRC code base.

## Scope

The function of the xBRC is described as the “block level”. The source code itself is not described in detail nor are the communication interfaces used to interact with other components. A separate, *xBRC Interface Control Document* is available to describe the latter.

## Assumptions and Constraints

It is assumed that the reader is familiar with the overall goal of the xBand system and the role that the xBRC plays in this system.

## Reference

|  |  |
| --- | --- |
| Document Number | Title and Revision |
|  | *xBRC Interface Control Document* |
|  | *xBR Interface Control Document* |
|  | *xFPE Interface Control Document* |

# Internal Design Notes

## Code Structure

The xBRC consists of a standalone Java application and a Tomcat web application. The former contains the actual xBRC event processing code whereas the latter implements the web-based user interface for configuring the xBRC.

The stand-alone Java application consists of */usr/share/xbrc/xbrc.jar* and all of the additional libraries stored in */usr/share/xbrc/lib*. The application is executed, indirectly, by the Linux daemon script in */etc/init.d/xbrc*.

The web user interface consists of the code and resources in */var/lib/tomcat6/webapps/UI.war*. It is implemented as a Java Struts web application.

In addition to the code components, the xBRC also makes extensive use of a MySQL database entitled *Mayhem*.

## Major Components

The diagram in the following section (2.3) illustrates the major components of the xBRC. These components are divided into 4 broad categories:

* The *event pipeline* – these components are involved in the processing of tap and long-range reader events, ultimately, translating them, as necessary into higher-level messages sent to downstream applications.
* The *model* – the xBRC can be configured to operate using different *modeling* code. A *model* (a particular Java class) is selected in the configuration file and the xBRC then sends events to that model for further processing. While the xBRC is use-case “blind”, the models are particular to specific use cases. There are unique models for park entry, for park attractions and for generic spaces. The *space* model is used in the Disney Cruise Lines use case and in the restaurant use case.
* *Housekeeping* components – the components in this category represent code that is needed for configuration and other purposes that is not directly involved in the event pipeline.
* *Web user interface* – the xBRC consists of a standalone Java application and a separate Tomcat web application. The event processing code, the model and the housekeeping components reside in the former while the web user interface is in the latter.

This organization of components is based on logical relationships rather than physical code structure. For example, the *web server* is included in the housekeeping components but it is an integral part of the event pipeline. Similarly, some *performance metrics* and *SNMP* status are collected during event processing but are also listed in the housekeeping section.

The sections that follow the component diagram describe each component in more detail and suggest ways that it can be tested.

## xBRC Major Components

  
Figure 1- xBRC Major Components

## Establishing Reader-to-xBRC Communication - the Reader Management Component

Before the event pipeline can function the dialog between readers and an xBRC must be established. The code to perform this is termed the *reader management* component. Note that the reader management component is embedded in the *web server* code and is not distinct. It begins with the *handleReaderPutHello* function and proceeds from there. Reader management relies on the following sequence:

The xBRC must be made aware of the reader. This can be done in various ways (e.g. using the *Location Editor*, establish a configuration with the xBRMS, running a database script) but whatever technique is used, it ultimately results in the proper configuration of the *Reader* table in the xBRC’s database. Most importantly, the MAC address and port entries in the database must be set up correctly as well as a *location id*.

A reader must be made aware of the xBRC. Mostly, this is done through DHCP configuration. When a reader requests a DHCP address, the DHCP server can be programmed to send that particular reader (as recognized by its MAC address) the IP address of the xBRC that it should communicate with. Alternatively, however, (at least, xFPE) readers can be manually configured (using an SSH session) with the address of their corresponding xBRC.  
  
To manually configure an xFPE, you can connect its network directly to a Windows or Linux computer then use *putty* or *ssh* to connect to the xFPE on the default IP address:  
  
 ssh root@169.254.0.2  
  
(No password will be needed). Once logged into the xFPE, you can use *vi* to edit the */mayhem/config.json* file and change the “xbrc url” property value to refer to a particular xBRC IP address and port.

The reader must send a PUT /hello message to the xBRC. The payload of this message (as described in the xBRC Interface Control Document), in JSON format, identifies the MAC address, TCP port and other aspects of the reader. Note that the reader will repeatedly send out /hello messages when trying to register itself with an xBRC. Even after successfully registering, it will still send out period /hello messages.

The xBRC must recognize the MAC address in the JSON payload of the hello message as having been previously configured (in #1). Note that if the xBRC does not recognized the reader, it will continue to interact with (as per the following steps), but it will not process any events coming from the reader beyond the “EKG Logging” step.

If necessary (if the xBRC has been configured to use a different “name” for the reader other than what it reports in the “hello” message), the xBRC sends a POST /reader/name message to the reader in order to rename it.

The xBRC puts the reader in “push” mode by performing a POST /update\_stream. This HTTP message includes the xBRCs “push URL” as well as parameters controlling how frequently data should be sent and how much old, cached, data should be transmitted.

The xBRC sets the reader’s clock to its own time by performing a POST /time.

The xBRC synchronizes the reader’s clock by performing a POST /time operation.

## The Event Pipeline

The event pipeline describes the code that is involved in receiving events from readers, processing them and, ultimately, sending higher-level messages to downstream applications. The code for the event pipeline is in the Process module and, spefically, in the *doProcess* function.

### Receiving Events – the *Reader Communication* Component

The code for receiving reader events from a particular reader is also embedded in the web server code. It begins with the function *handleReaderPutUpdateStream.* Conceptually, this function is very simple:

* Deserialize events from the PUT /stream message (from JSON into Java objects).
* Push the events into the appropriate buffer (see *Double Buffering* below)
* Keep track of the last event id received from the reader (store it in the database)

### Double Buffering

When readers PUT data to the xBRC, the xBRC processes these messages using web server threads. The use of multiple threads allows the web server to service many readers in an efficient fashion. The events PUT by the readers are collected in a buffer. Periodically, the xBRC “switches” to a second buffer while it processes the events collected in the first. This use of double buffering minimizes multi-threaded contention issues and results in superior performance.

The double buffering code is implemented in the *Processor* Java module. The *pushEvents* function is the one called by the reader communication component (namely, the *handleReaderPutUpdateStream* function). It pushes reader events into the current “from reader” buffer while other code in the event pipeline processes events in the other buffer. The code in the *preProcessing* function swaps between the two buffers each time it is called.

### Data Slice Detection

The data slice detection code is also present in the *preprocessing* function. After swapping buffers, it sorts the data in the current “being processed” buffer (by event timestamp) then looks to see if any long-range messages have been received “too recently”. If they have, there is the possibility that other messages will soon be received (or have already been received, but are in the other buffer) that should be processed coincidentally with the current ones. Any such messages are *deferred* by being pushed “back” into the “from reader” buffer to be reconsidered in the next iteration.

### ID Lookup

Events from xFPE readers identify the RFID (tap id) of the band that tapped the device. Events from xBR readers identify the LRID (long-range id) of the band that communicated with the reader. In order for the xBRC to perform some of its functions, it needs to associate these ids with a particular *guest*. The xBRC performs this operation indirectly while writing to its “EKG” log file. The *logToEKGFile* in the *Process* module calls *Xview.getInstance().getBandIDFromLRID* or *Xview.getInstance().getBandIDFromRFID* to first map an LRID or an RFID to a band id. It then calls *Xview.getInstance().getGuestIDFromBandId* to determine what guest has been assigned that band. Note that the *Xview* calls ultimately communicate with IDMS using HTTP RESTful mechanisms. Once these *Xview* functions have been called, the information remains cached and available for subsequent calls. Note, too, that the data returned by the HTTP RESTful call for *getBandIDFromLRID* or *getBandIDFromRFID* contains information about the guest (if any) to which the band has been assigned. When the subsequent call to *getGuestIDFromBandId* is made, no HTTP operation is performed as the guest information has already been cached.

### Logging Events to the “EKG” File

Low-level events are logged to what is termed the “EKG” file (because monitoring it tells you if the xBRC is “alive” or not). This file is typically sent to */var/log/xbrc/eventdump.txt*.

Because this file can be read over HTTP (described later), it is written and read using a random-access file. The *logToEKGFile* performs the output formatting whereas the Java Log module performs the actual output.

The general format of the EKG file is:  
  
 <time stamp>,<record type>,<payload>

For example:

1324574032289,PROCESS,58

1324574032216,TAP,xpassentry,Bethany Bell,7058b33531610b0d

1324574032217,LRR,entry-4,Jill Neusch,38bb1a5a5dd5b83d,0,-52,2476,1

1324574032217,LRR,entry-4,Jill Neusch,38bb1a5a5dd5b83d,0,-52,2476,0

1324574032217,LRR,entry-4,Cedric Douglas,2a12e9ef5d4b252a,0,-45,2401,0

1324574032217,LRR,entry-4,Cedric Douglas,2a12e9ef5d4b252a,0,-45,2401,1

1324574032217,LRR,entry-4,Theodore Dauila,3d85302178ba59d6,0,-56,2476,0

1324574032217,LRR,entry-4,Theodore Dauila,3d85302178ba59d6,0,-56,2476,1

1324574032217,LRR,entry-3,Cedric Douglas,2a12e9ef5d4b252a,0,-55,2401,0

1324574032217,LRR,entry-3,Cedric Douglas,2a12e9ef5d4b252a,0,-55,2401,1

1324574032217,SNG,Entry,2a12e9ef5d4b252a,Only

1324574032217,SNG,Entry,38bb1a5a5dd5b83d,Only

1324574032217,SNG,Entry,3d85302178ba59d6,Only

<time stamp> is in the form of “epoch millisecond time” (number of milliseconds after January 1st, 1970 00:00:00 UTC. <payload> varies according to the <record type>. The different record types are:

PROCESS – indicates how many reader events are being processed by the xBRC in the current batch.

TAP – indicates an xFPE tap event. The payload describes the name of the reader that reported the event, the name of the guest associated with the tapped band and the RFID (tap id) of the band.

LRR – indicates an xBR long range read event. The payload describes the name of the reader that reported the event, the name of the guest associated with the transmitting band, the long-range id of the band, the transmission’s “packet number” (cycles from 0-255 repeatedly), the signal strength in dB (typically -90 to -40), the frequency and the radio “bank” (0 or 1) that picked up the transmission in the xBR.

SNG – indicates an xBRC singulation. The payload identifies the reader location where the a band has been determined to be, the long-range id of the band and the “basis” for the singulation. “Only” indicates that only one reader location was reporting data for the guest. Other bases included “MeanSS”, “PeakSS”, “Count” and “First”.

DEFER – indicates that a number of events have been deferred for later processing. The payload identifies the count of such events. Event deferral currently occurs only for long-range events when the xBRC heuristically determines that more related events may be coming “soon.”

### Singulation

After logging, the event pipeline now performs its analytical functions. These are performed in various steps, driven by the *performModelProcessing* function:

The selected model is called (*beforeProcessEvents*) to allow any pre-processing.

The events are *singulated*

The model is called to process the singulated events (*processEvents*)

The model is called (*afterProcessEvents*) to perform any post processing

Event singulation currently consists of determining what *location* a band is closest to as determined by analyzing reader events. This determination is done by the *EventAggregator* Java module. This module’s *Analyze* function is given a list of reader events (the “being processed” buffer) and it returns a list of “aggregated” events. If multiple long-range events (the same band “chirp” being received by multiple channels or multiple readers in the same long-range reader location) suggest that a band is at a particular long-range reader location only a single event aggregate object is returned for those events. Tap and other events don’t require singulation and are simply converted into analogous aggregated events (*TapEventAggregate*, *BioEventAggregate*, etc.).

Note, too, that the *EventAggregator* can be configured to use different singulation algorithms. Currently, two algorithms are supported: “max” or “mean” (as selected by the *ControllerInfo.singulationalgorithm* configuration setting). The former algorithm, when comparing events from multiple locations, selects the location which reports the largest signal strength. The latter algorithm selects the location with the largest mean signal strength as calculated using the data from reporting readers.

The results of the singulation step are recorded in the EKG file. The *basis* field describes the basis for the determination of the algorithm:

|  |  |
| --- | --- |
| Basis | Description |
| Only | Only one reader location is reporting any events for the band. |
| MaxSS | The singulation was based on which location reported the maximum signal strength for a band transmission. |
| MeanSS | The singulation was based on which location reported the largest mean signal strength for the readers reporting the event. |
| Count | The singulation was based on the count of radios reporting an event. |
| First | As all other metrics were equal, the singulation was based on the first reader location reporting an event. |

### Modeling

The modeling step is actually three steps: *beforeProcessEvents, processEvents* and *afterProcessEvents*. Frequently, the first and third step are bookkeeping functions and the bulk of the actual “modeling” takes place in *processEvents*.

The details of what *processEvents* does is specific to the type of model that is being used. The *attraction* model, for example, implements a state machine that keeps track of guest state (“entered”, “merged”, “loaded”, “in car”, and “exited”) sending high-level messages on each state transition and calculating various wait times, as well. Additionally, the attraction model also aggregates various operation metrics over time (for example, average xPass wait time) and sends high-level metrics messages with this data. The *space* model, on the other hand, only keeps track of a guest’s location and sends high-level messages whenever a guest changes locations or whenever a guest “abandons” a location. The *park entry* model, on the other hand, maintains a much more complicated state machine that manages guests’ interactions with biometric readers and with entry entitlements.

Regardless of what a model does, its role in the event pipeline is the same: to interpret low-level messages and to interact with cooperating or downstream applications. Additionally, in order for the *Facility View* feature (of the Web UI) to work, the model must also maintain the GST (guest status table) in the database. The Facility View is, essentially, a visualization of how many guests are in specific states or in specific locations. The UI code determines this information by consulting the GST table in the database.

### Messaging

The xBRC supports both HTTP and JMS (using Progress Software’s SonicMQ) messaging. Models can generate messages without concerning themselves with how those messages will be delivered.

To send a message, a model creates an XBRCMessage object and calls *XBRCController.getInstance().getConnector().publishMessage*. Ultimately, this ends up in the *JMSAgent* Java module. Initially, all that happens is that the message gets serialized to a string and gets written to the *Messages* table in the database.

The JMSAgent module, however, also implements a “send thread” (see *runSendThread*). This thread wakes up periodically and sends any new messages which have been written to the database. It calls the *sendQueuedMessages* function which, in turn, calls either *sendQueuedMessagesViaHTTP* or *sendQueuedMessagesViaJMS*. Which path it takes is determined by the *ControllerInfo.updatestreamurl* configuration value. If this value is set to a valid URL, the messages will be sent over HTTP. Alternatively, if the value is not set but the *ESBInfo.jmsbroker* value **is** set then the messages will be sent over the JMS.

Note that if sending a message via the JMS fails, that the xBRC will continue trying to send messages and will send all queued messages if it ultimately succeeds in establish communication with the JMS broker. If the xBRC is configured to use HTTP, a failure to communicate will result in a cessation of subsequent communication until the downstream application reestablishes communication (by performing a PUT /update\_streamoperation as described in the *xBRC Interface Control Document*).

#### Fast delivery of LOCATIONEVENT messages for TAP events

Since the version 1.6 of the xBRC, the xBRC supports fast delivery of LOCATIONEVENT messages for TAP events. This mechanism is intended for systems that only require the public RFID for a TAP event and do not require other guest information.

The intention of this feature is to deliver the LOCATIONEVENT message for a TAP as quickly as possible to downstream application over HTTP REST call.

To enable this functionality set the “fasttapurl” Config property to the URL of the downstream application.

## Receiving raw Reader TAP or GPIO events

Since the DAP reader version xTP2-2.4.0-2654, the DAP reader supports sending of reader events to multiple streams. This means that a reader can send raw events to the xBRC while at the same time sending events to other systems. The xBRC is used to configure the URLs to which the reader will send events. To enable this functionality use the “readerupdatestreams” Config property, documented in the xBRC Configuration Values section later in this document.

The xBRC will set these URLs on the reader when the reader makes the HELLO REST call to the xBRC.

## Housekeeping Components

As with the previous section, the components in this section are not necessarily distinct groupings in the code. I describe them as such only to combine related functional items.

### Configuration Processing

xBRC “configuration” refers to various data which must be specified in order for an xBRC to function properly. These include:

* The location and credentials to be used when accessing the xBRC database. This is done in */etc/nge/config/environment.properties* and consists of several items:
  + The jdbc connection string for the database
  + The username and password to be used with the database
  + Configuration parameters for the database connection pool
* Information about the long-range and tap readers being used in the facility. This data is stored in the database (in the *Reader* table) and includes:
  + Reader MAC address and port
  + Reader name
  + Reader *location id* (index into separate Location table)
  + Reader type (long range, tap, bio)
* Information about the reader locations being used in the facility (see *location id*) above. This data is stored in the database (in the *Location* table) and includes:
  + Location id
  + Location name
  + Location type (entry, xpassentry, merge, load, exit, incar or waypoint)
* Operational control values. These include both model-independent and model-specific values, both stored in the Config table. A list of model-independent and model dependent values is included later in this document.
* *Facility designer* information stored in the database. The Facility Designer allows an installer to create the “subway diagram” for a facility – a high-level diagram that illustrates the readers and states relevant to the installation. This design is then show in the *facility view* user interface decorated with guest counts for various states and reader locations.

The */etc/nge/config/environment.properties* file is typically hand-edited as it must be configured before anything else in the xBRC is operational. Information about readers and reader locations can be set using the *location editor* in the xBRC Web User Interface. Alternatively, an XML configuration file can be POSTed to the xBRC using HTTP mechanisms and then selected as the current configuration.

Operational control values can be set using the xBRCs command-line interface or through an xBRMS.

Facility designer information can only be created interactively by using the xBRC Web user interface.

#### Stored Configurations

Note that the xBRC supports the concept of *stored* configurations. The current configuration can be saved to the database using HTTP PUT /storeconfiguration. New configurations can be uploaded by using HTTP POST /newconfiguration and specifying the new configuration (in XML form) in the HTTP payload. The current configuration can be examined with HTTP GET /currentconfiguration (note: the output of GET /currentconfiguration can then be used as a template for new, uploaded, configurations). Configurations can be enumerated with HTTP GET /configurations and a stored configuration can be selected as the current one by using HTTP PUT /selectconfiguration.

Stored configurations can be either “full” or “incremental”. A full configuration completely replaces the current configuration when selected while an incremental configuration only overwrites the information that it provides. Note that the only way to create an incremental configuration is by using HTTP POST /newconfiguration.

#### Run-time Updates of Configuration Information

The xBRC can dynamically re-read and reapply configuration information while it is executing. The xBRC will do this in response to an HTTP PUT /updateconfig operation. Note that some configuration values (for example, database location, database connection pool values, xBRC model, event dump file location, etc.) cannot be changed dynamically.

The xBRC command-line interface automatically performs this PUT operation whenever it is used to modify a configuration value (unless the --noupdate flag is specified). The xBRMS and Web User Interface are still in the process of being modified to exploit this functionality.

### Command-Line User Interface

The */usr/share/xbrc/xbrcconfig* program provides a simple user interface for performing various operations:

* Creating and testing the presence of the xBRC database
* Listing, modifying and removing operational control values
* Informing the xBRC that it should re-read its configuration information from the database.

|  |  |
| --- | --- |
| Command-Line | *Description* |
| xbrcconfig --createdb | Create the xBRC databases and runs run the necessary SQL scripts in */usr/share/xbrc/dbscripts* directory to bring the databse the the latest version. Database operations are performed using the root user and password specified in the /etc/nge/config/environment.properties file. |
| xbrcconfig --testdb | Verify that the xBRC database has been built and is accessible. |
| xbrcconfig --list | List all of the configuration values. |
| xbrcconfig CLASS.PROPERTY=VALUE | Add or modify a configuration value. The xBRC will automatically be told to re-read its configuration values unless --noupdate is specified. Refer to section 5 for details on different CLASS and PROPERTY settings. |
| xbrcconfig --delete:CLASS.PROPERTY | Removes a configuration parameter. The xBRC will automatically be told to re-read its configuration values unless --noupdate is specified. |
| xbrcconfig --updateconfig | Tells the xBRC to re-read configuration values. This command is frequently used after a series of values have been changed using the --noupdate option. |
| xbrcconfig --encrypt <key> <text> | Encrypt some text using the specified key. The encrypted text is printed out to the console and may be used as a property value in the environment.properties file by enclosing it in ENC() (e.g. property=ENC(AxZfdklkzjdlfakdj).) |

Any command which can result in modification or deletion of data will prompt for confirmation. This prompt can be suppressed by specifying *--force* with the operation.

Note that running the *xbrcconfig* program requires administrator privileges.

### Performance Metrics

The xBRC calculates several performance metrics and includes their values as part of its XML response to HTTP GET /status. For each of the metrics below, the /status output includes the minimum, maximum and mean value observed during the current sampling interval:

|  |  |
| --- | --- |
| Metric | Description |
| Events | Number of events in the incoming double buffer queue |
| EventAge | The age of the oldest event in the queue |
| IDMSQuery | Amount of time (in msec) spent, per event, talking to the IDMS |
| EKGWrite | Amount of time (in msec) spent, per event, writing to the EKG file |
| Singulation | Amount of time (in msec) spent, per event, singulating |
| PreModeling | Amount of time (in msec) spent in the pre-modeling step |
| Modeling | Amount of time (in msec), per aggregated event, spent in the modeling step |
| PostModeling | Amount of time (in msec) spent in the post-modeling step |
| External | Amount of time (in msec) spent waiting for external systems (e.g. GXP, Omni, etc.) |
| WriteToReader | Amount of time (in msec) spent sending information to readers |
| SaveGST | Amount of time (in msec) spent writing the GST to the database |
| Upstream | Amount of time (in msec) spent sending messages (via HTTP or JMS) to other applications |
| MainLoopUtilization | Percentage of the main loops “quantum” that was needed. Over 100% indicates that the xBRC is falling behind on its processing |
| Model1 | Model-specific datum |
| Model2 | Model-specific datum |
| Model3 | Model-specific datum |

The min/max/mean data is calculated over a configured sampling interval (*ControllerInfo.perfmetricsperiod*) specified in seconds.

The output of HTTP GET /status indicates when (at what UTC time) the current sample began (with the <startPerfTime> tag) as well as the period (<perfMetricsPeriod>). A calling program should use these values to sample (repeatedly call HTTP GET /status) near the end of each sampling period in order to obtain the most useful results.

### Web Server

The xBRC incorporates a multi-threaded open source server package: simple-4.1.21.jar (note: it has a non-restrictive license: <http://www.apache.org/licenses/LICENSE-2.0>). This code is reputed to be fast and a reliable. We have not had any problems with it.

The web server instantiated in the xBRC WebServer Java module and initialized in the *Initialize* function. The web server can operate over HTTP and over HTTPS depending on whether both the *ControllerInfo.httpport* and *ControllerInfo.httpsport* configuration values have been set. If the HTTPS port has been set, then all non-reader HTTP requests **must** be received over the HTTPS port. In other words, the specification of a value for the HTTPS port implies that all non-reader requests must be encrypted.

### Web Request Processing

The processing of web server requests also occurs in the WebServer Java module. The xBRC responds to a variety of GET, PUT, POST and DELETE operations as described in the xBRC Interface Control Document. In general, these operations will return a status code of 200 for successful operations, a status code of 404 if an improper path or some requested datum does not exist and a status code of 500 if an exception occurs. Error values are accompanied by HTTP payloads that describe the cause of the error (not all browsers will display these).

### Daemon Processing

The xBRC is designed to run as a Linux daemon. An /etc/init.d script is provided so that it can be started as such. Additionally, the xBRC RPM installation script will perform the necessary “chkconfig” operation to assure that the xBRC starts up automatically and in the correct sequence (after MySql).

The /etc/init.d script supports “start”, “stop” and “restart”.

Note, too, that the /etc/init.d script uses the *jsvc* program to start up the xBRC and to invoke *setuid* so that it runs as *xbrcuser* instead of as *root*.

The xBRC sends log file output to */var/log/xbrc*.

### SNMP

TBD (need to get information from Iwona).

### The Scheduler

The xBRC has a built-in scheduler. The scheduler is capable of executing various pre-defined tasks.

A Scheduler Task defines a job that is executed by the xBRC scheduler according to the schedule defined by the crontab expression.

Each scheduler task type is pre-defined by the xBRC. It consists of a Java class that runs the task and a set of parameters that control the task execution. Multiple tasks of the same type may be created, but care must be taken to ensure that these tasks will not run concurrently, unless a task does allow for concurrent execution. For example, any database cleanup tasks should not be scheduled to run concurrently.

The xBRC UI Scheduler Tasks page allows for creating, deleting tasks and modifying their parameters. This includes modifying the task schedule which is expressed as a crontab expression.

# Model-Specific Information

The modeling operations described in section 2.5.7 and the messages sent out via the techniques in section 2.5.8 vary depending on which model is in effect for a particular xBRC installation. This section describes the three models currently defined (attraction, park entry and space).

## Attraction Model

This model is used for park “rides”. Although each attraction is different, most will share several traits:

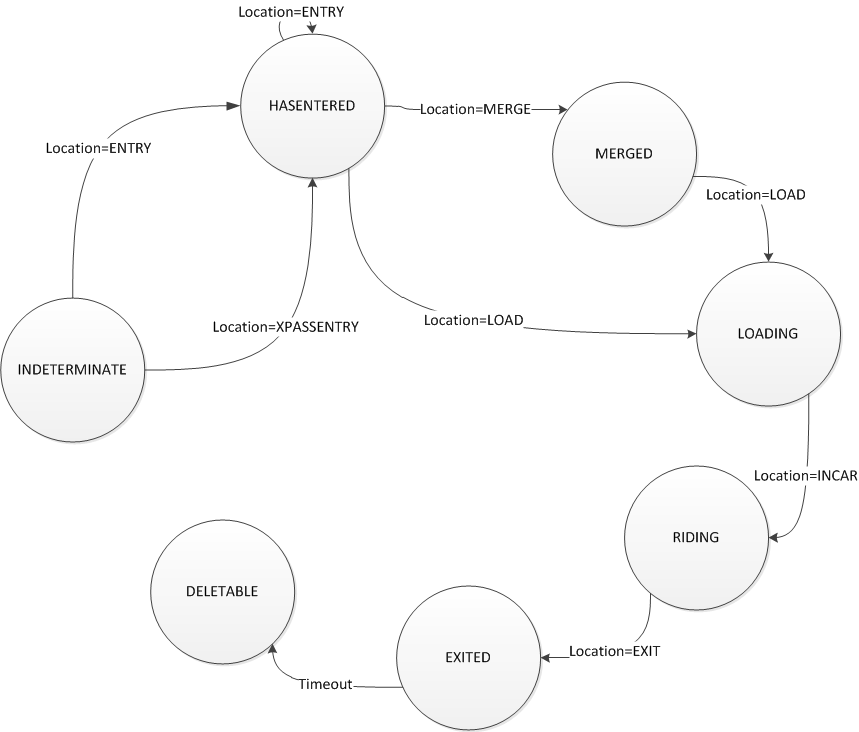
* A standby guest queue with a long-range reader location at entry
* An xPass queue with a tap reader at entry and another at merge (although only a single “combo” reader may be present)
* A long-range reader location near the standby/xPass queue merge area
* One or more long-range reader location near the attraction load point
* A long-range reader location near the attraction exit

An attraction might have a long-range reader location somewhere along the ride path indicating that any nearby guests are currently “riding” an attraction vehicle.

An attraction might also have one or more “waypoint” locations (either long-range or tap) that are used to trigger interactive experiences.

### State Diagram

The attraction model processes these different reader location types in order to maintain a state machine for each guest. Here is a simplified version of this machine:

  
Figure - Attraction Model State Machine

The state machine facilitates the determination of where a guest is and how his/her progress through the attraction should be messaged to downstream applications. In general, a guest “advances” from the “HASENTERED” state to “MERGED” then to “LOADING”, etc. until he or she reaches the “EXITED” and then the “DELETABLE” state. Guests are not allowed to “back up” to previous states. Once a guest is sensed at a load reader, for example, that guest is placed in “LOADING” state and will not return to “HASENTERED” state even if he or she is sensed at an entry xBR.

When progressing through the state machine, the model logic is tolerant of guests that may have been “missed” at certain reader locations. For example, a guest in the “HASENTERED” state that shows up at an exit reader is placed in “EXITED” state; the model assumes that the guest was missed by intermediate reader locations.

### Messaging

Generally, the attraction model generates messages whenever guests transition from one state to another. Note that the message for the “HASENTERED” state may be “deferred” if the *AttractionModelConfig.deferentrymessages* configuration value is set to “true”. Deferring these messages reduces “noise” from guests that might be singulated to an entrance reader location but that might not actually enter an attraction. A message is also sent if a guest “disappears” (stops sending any events) from the attraction (except when a guest in the “HASENTERED” state disappears and *deferentrymessages* is set to “true”).

A message is also generated whenever a guest is first sensed at a waypoint location although no state transition occurs. No message is generated when a guest becomes “DELETABLE”.

The format for messages is detailed in the *xBRC Interface Control Document*.

### Entitlement Redemption

When guests tap readers at xPass entry or xPass merge locations, the attraction model communicates with an external GXP server in order to determine whether guests have the proper entitlement to pass through the xPass queue. The *AttractionModelConfig.gxpurl* configuration value specifies the location of this server. The *xBRC Interface Control Document* describes the transactions which the xBRC performs with the GXP server. The result of these transactions is a message being sent back to the originating tap reader telling it to turn its green or blue light on to indicate success or failure. Additional information is provided to attraction cast members by the GXP system without xBRC interaction.

### Vehicle Association using the AGC event

For those attractions equipped with vehicles it is desirable to associate guests to the vehicle in which they are traveling.

Refer to the “xBRC Guest to Vehicle Association.docx” document for detailed explanation of vehicle association.

## Space Model

The space model is used by the DCL (cruise line) applications and by the park restaurant use cases. This model is much simpler than the attraction model - little guest state is maintained. When a guest is first singulated to a reader location, a READEREVENT message is sent (as described in the *xBRC Interface Control Document*). As long as the guest remains in that location, no additional messages are sent. If a guest leaves the location (without appearing at a different location), an ABANDON message is sent. Any tap events are sent as READEREVENT messages, as well.

## Park Entry Model

**NOTE: Refer to the xBRC Park Entry Model document for detailed park entry model documentation.**

The park entry model is used at the entrance to the park. The single task of this model is to validate a park pass for each guest allowing the guest to enter the park. The physical setup of park entry includes a series of tap readers equipped with biometric finger scanners (xfp+xbio). Multiple readers are grouped under one or more named locations. A typical guest interaction is as follows. Guest approaches a tap reader; taps the park RFID-equipped pass on the reader; waits for the biometric scanner light; places a finger on the scanner; waits for a green light and enters the park. Other guest interaction scenarios are documented later on in this section.

### Omni Ticketing System

The entitlement information for each guest is maintained by the Disney’s ticketing system called Omni. The park entry model communicates with the Omni system via XML messages send over TCP/IP network protocol. The Omni system also maintains biometric finger information for each guest obtained during the enrolment event as each guest enters the park for the first time.

Omni requires that all xfp+xbio readers are in the connected and logged-in state before they are used to process guests. The park entry xBRC sends a “connect” XML message for each reader that is alive and broadcasting “hello” messages. Furthermore, the xBRC periodically sends a “watchdog” XML message to Omni for each reader. A cast member is responsible for a group of readers. Prior to processing guests, a cast member logs into each reader location which in turn logs each reader into the Omni system using this cast members’ Omni credentials.

# Web User Interface

The xBRC web user interface is not intended for day-to-day use. It is, however, valuable when first configuring an xBRC for a particular installation and to verify its basic operation. The user interface (accessible through http://<xbrc address>:8090/UI) consists of four initial pages, one of which leads to another subsequent page:

|  |  |
| --- | --- |
| Page | Description |
| Marching Guests | This page is “deprecated”. It is used to visualize the x,y locations of simulated guests and has no benefit in a production level xBRC. |
| Facility Designer | This page implements an interactive, drag-and-drop, interface that allows a facility installer to design the “subway diagram” for the facility. The subway diagram depicts the various reader locations present in the facility and the flow of guests between them. |
| Facility View | This page displays the subway diagram annotated with guest counts determined by analyzing the GST (Guest Status Table) for the xBRC. |
| Location Editor | This page allows a facility designer to specify the reader locations present in a facility and to assign readers to these locations. |

The *Facility View* page allows the user to click on a displayed guest count in order to bring up a popup window that displays the names of the guests included in that count (for example, the guests in the “Entry” location). The user is then allowed to click on the “magnifying glass” displayed for a particular guest in order to bring up the “Signal Strength” page (Note: this isn’t the actual title).

# xBRC Configuration Values

## *ControllerInfo* Settings

*ControllerInfo* values are model-independent. These values are respected by the core xBRC code rather than by specific models. When setting these values via *xbrcconfig* or other means, the “class” should be specified as “ControllerInfo”.

|  |  |  |
| --- | --- | --- |
| Property | Example Value | Description |
| eventdumpfile | # | The path for the EKG file. No file will be written if this starts with “#” |
| eventprocessingperiod\_msec | 200 | How often the event pipeline should be serviced |
| httpport | 8080 | Which port the xBRC should listen to |
| httpsport | 0 | Which SSL port the xBRC should listen to (leave at 0 if none) |
| idmscachetime\_sec | 1800 | How long (seconds) IDMS results should be cached |
| jmsretryperiod | 1000 | How frequently (msec) should attempts be made to establish connection with a faulty JMS broker |
| maxreadsbeforepurge | 32 | How many long-range read samples should be stored for a “watched” band (supports the power-level display) |
| messagestorageperiod | 3600 | How long (in seconds) should messages be stored by the xBRC before being discarded (whether sent or not) |
| model | com.disney.xband.xbrc.attractionmodel.CEP | The class name of the model to use |
| ownipprefix | 10. | If the xBRC has multiple NICs, which NIC IP prefix should be considered the “preferred” interface |
| perfmetricsperiod | 15 | How long (in seconds) should the performance metrics period be |
| readerdatasendperiod\_msec | 100 | How frequently (in msec) should readers be told to send data |
| readerhellotimeoutsec | 90 | How long (in seconds) should the xBRC wait for a /hello message before it considers a reader to be broken |
| readerhttptimeout\_msec | 3000 | How long (in msec) should the xBRC wait for HTTP responses from readers |
| reducedataslicing | true | Set to “true” to enable the algorithm that abates data slicing |
| reversetapids | false | Set to “true” to reverse RFIDs received from xFPE readers |
| setreadertime | true | Set to “true” if the xBRC should synchronize reader clocks |
| singulationalgorithm | max | Set to “max” or “mean” accoriding to which singulation algorithm is preferred |
| updatestreamhttptimeout\_msec | 2000 | How long (in msec) to wait for a response to a message being sent via HTTP to the update stream |
| updatestreamurl | # | The URL for an HTTP (vs JMS) message destination |
| updatestreamfasttaps | false | If set to true then LOCATIONEVENT messages for TAP events sent to the updatestreamurl will be fast forwarded bypassing IDMS band to guest resolution. |
| venue | 80010114 | The OneSource provided “name” for the facility |
| verbose | false | Set to “true” to generate additional log output (poorly supported) |
| watchedbandtimeout | 180 | How long (in seconds) to keep information about a watched band if no new reader data is received for it |
| xviewurl | http://localhost:8090/Xview | The URL of the IDMS or xView server that should be used for ID resolution |
| fasttapurl |  | The URL to send LOCATIONEVENT messages for TAP events bypassing IDMS band to guest resolution. This guarantees very fast delivery of TAP messages, within 20-30 millisecons. |
| readerupdatestreams |  | Pipe separated list of URLs where the DAP readers will post raw TAP events to. This allows for a system other than the xBRC to receive raw TAP events from a reader. |
| diskspacemonitorpath | / | Which disk path to monitor for low disk space |
| diskspacewarnmb | 500 (MB) | Return a warning status message if disk space falls below this limit |
| diskspacefatalmb | 200 (MB) | Return a fatal status message if disk space falls below this limit |

## *ESBInfo* Settings

*ESBInfo* values are also model-independent. These values are respected by the core xBRC code when communicating with the Sonic MQ JMS. When setting these values via *xbrcconfig* or other means, the “class” should be specified as “ESBInfo”.

|  |  |  |
| --- | --- | --- |
| Property | Example Value | Description |
| jmsbroker | 10.75.2.56:2506 | Address and port of JMS broker |
| jmsdiscoverytimesec | 60 | How frequently (in seconds) to send JMS discovery messages to the xBRMS |
| jmspassword | Administrator | The password to be used when talking to the JMS broker |
| jmstopic | com.synapse.xbrc | The topic to be used for publish/subscribe by the xBRC |
| jmsuser | Administrator | The username to be used when talking to the JMS broker |

## Model-SpecificSettings

Each model can maintain its own model-specific properties in the configuration database. The following values are relevant only to the attraction model. When setting these values via *xbrcconfig* or other means, the “class” should be specified as “AttractionModelConfig”.

|  |  |  |
| --- | --- | --- |
| Property | Example Value | Description |
| abandonmenttimeout | 60 | How long (in seconds) to wait before a guest that is not appearing at any reader should be considered to have “abandoned” the attraction |
| bluelightduration | 2500 | How long (msec) to light the blue light on an xFPE |
| deferentrymessages | true | “true” if entry messages should be deferred until a guest appears at a subsequent reader |
| exitstatetimeout | 120 | How long a guest should be kept in “exit” state before being deleted from the GST |
| greenlightduration | 1000 | How long (msec) to light the green light on the xFPE |
| gstsaveperiod | 1000 | How frequently (msec) the GST should be saved to the database |
| gxpurl | # | The URL to be used for xPass entitlement checking. No checking performed if this starts with “#” |
| metricsperiod | 15 | How often (in seconds) should model metrics be sent |

TBD - settings for space model and park entry

# Installing the xBRC Code

## Installing and Updating xBRC Code

The xBRC is installed using standard Red Hat/Centos “rpm” package files. To install the code, you select the package that corresponds to the *model* relevant to your use case:

|  |
| --- |
| Package |
| xbrc-attraction-1.0.0.*BUILD*.i386.rpm |
| xbrc-parkentry-1.0.0.*BUILD*.i386.rpm |
| xbrc-space-1.0.0.*BUILD*.i386.rpm |

*“BUILD*” is a placeholder for a particular xBRC build number.

While the customer will typically install an xBRC package using some type of automatic deployment mechanism, you can manually install it using the *rpm* command in Linux, for example:

sudo rpm -i xbrc-attraction-1.0.0.747.i386.rpm

You can determine which package is currently installed with:

sudo rpm -qa | grep xbrc

If any xBRC package *is* installed, you can update it with a more current version by using:

sudo rpm -U xbrc-attraction-1.0.0.748.i386.rpm

## Location of xBRC Files and Directories

The xBRC rpms install a variety of files in different locations:

|  |  |
| --- | --- |
| File | Description |
| /usr/share/xbrc/xbrc.jar | The Java jar file that contains the main xBRC entry point |
| /usr/share/xbrc/xbrcconfig | Command-line user interface for the xBRC |
| /etc/nge/config/ environment.properties | Property file that, minimally, tells the xBRC connection information for its database |
| /usr/share/xbrc/log4j.xml | Configuration file that controls where and how verbosely the xBRC logs diagnostic information |
| /usr/share/xbrc/initdb.sql | MySQL script, used by the “xbrcconfig --createdb” command. Creates a bare-bones database specific to the type of model in use. |
| /usr/share/xbrc/jsvc-xbrc | Bootstrap application used by /etc/init.d/xbrc daemon script. Allows xBRC to run as non-root user |
| /usr/share/xbrc/xbrc.keys | Certificate store that contains SSL server key for xBRC (when using SSL communication) |
| /usr/share/xbrc/lib/\* | Libraries (additional Java jar files) needed by the xBRC |
| /etc/init.d/xbrc | Standard Linux daemon script for xBRC |
| /etc/logrotate.d/xbrc | Log rotation script to rotate /var/lib/xbrc/eventdump.txt |
| /var/lib/tomcat6/UI.war | xBRC web interface code and resources |
| /var/lib/tomcat6/Xview.war | (Temporary - will be removed when we work exclusively with IDMS) |

The configuration and script files for the xBRC (namely, /usr/share/xbrc/log4j.xml and /etc/init.d/xbrc) result in files being created in other directories:

|  |  |
| --- | --- |
| File | Description |
| /var/log/xbrc/xbrcController.log | The main log file for the xBRC. |
| /var/log/xbrc/xbrcInit.log | Log file used while starting xBRC daemon |
| /var/log/xbrc/eventdump.txt | The “EKG” log file. |
| /var/log/xbrc/  xbrcControllerPerformance.log | Used for logging timing information to gauge performance (typically, disabled). |
| /var/run/xbrc/xbrc/xbrc.pid | Contains the *pid* of the xBRC daemon process when it is running. |

## Creating the xBRC Database and Configuring /etc/nge/config/environment.properties

Before the xBRC can be started, its database must be created and the connection information for the database established.

The xBRC, when installed using an RPM package will attempt to automatically create or upgrade the xBRC (Mayhem) Database as the last step of the installation step. This is accomplished with the aid of the command “/usr/share/xbrc/xbrcconfig --createdb” executed from the RPM script.

### Automatic Database Creation and Upgrade

The command /usr/share/xbrc/xbrcconfig --createdb examines the current state of the xBRC database and performs any necessary steps to bring the xBRC database to the latest required version. This may include a creation of the Mayhem database, creation of the database user with appropriate rights to access the database as well as creation of the database schema and any seed data required. The database schema scripts are located in the /usr/share/xbrc/dbscripts directory and are installed by the xBRC RPM package.

The */etc/nge/config/environment.properties* file tells the xBRC where and how to find its database. The properties from this file are also used to create the xBRC database for the first time. Properties with sensitive information such as passwords may be encrypted. The encrypted entries are sourrounded with ENC(). Plaintext may be encrypted by running the following command.

/usr/share/xbrc/xbrcconfig --encrypt <key> <text>

Figure - */etc/nge/config/environment.properties*

nge.xconnect.xbrc.dbserver.service=jdbc:mysql://localhost/Mayhem  
nge.xconnect.xbrc.dbserver.uid=EMUser  
nge.xconnect.xbrc.dbserver.pwd=Mayhem!23   
nge.xconnect.xbrc.dbserver.rootuid=root  
nge.xconnect.xbrc.dbserver.rootpwd=  
nge.xconnect.xbrc.dbserver.newrootpwd=Mayhem!24   
nge.xconnect.xbrc.c3p0.maxPoolSize=20  
nge.xconnect.xbrc.c3p0.maxStatements=400  
nge.xconnect.xbrc.c3p0.maxStatementsPerConnection=50

The encrypted entries (e.g. nge.xconnect.xbrc.dbserver.pwd=ENC(/33AubBVTTbk7aqMYbN59hWESdiCKTtUpni5NU9TeP8=)) are automatically decrypted when the xBRC reads the /etc/nge/config/environment.properties file. The key used to decrypt the file is read from /etc/nge/config/jasypt.properties.

nge.jasypt.xconnect.key=disney

During the automatic database creation the scripts are executed as the user specified by nge.xconnect.xbrc.dbserver.rootuid. The nge.xconnect.xbrc.dbserver.rootpwd must be the current password for that user. If the nge.xconnect.xbrc.dbserver.newrootpwd parameter is set to a new password then it will be automatically changed in the MySQL server to this password.

The *c3p0* properties control the database connection pool software and should generally be specified as per the figure.

### Starting the xBRC

Once the database properties have been set, the xBRC can be started:

/etc/init.d/xbrc start

### Setting Additional Configuration Values

It may be valuable to set some additional configuration values before proceeding further. For example:

cd /usr/share/xbrcconfig  
 sudo ./xbrcconfig ESBInfo.jmsbroker=10.75.2.56:2506  
 sudo ./xbrcconfig ESBInfo.jmsuser=Administrator  
 sudo ./xbrcconfig ESBInfo.jmspassword=Password

This sets the JMS broker address to 10.75.2.56, the JMS username to “Administrator” and the password to “Password”.

Setting the JMS broker not only tells the xBRC that it should communicate through JMS, it also allows an listening xBRMS to automatically detect the xBRC and to start monitoring it. Once an xBRMS is aware of an xBRC, it can also be used to remotely modify configuration settings on the xBRC.