

openLuup

@akboorer, August 2016

Contents

openLuup	1
Contents	1
Overview	2
2016, Release 8.5 (v16.8.5)	3
Installation	4
1. Lua installation	4
2. openLuup Installation	5
3. Linking to Vera	7
Backing up the system	8
More about VeraBridge	9
1. Bridging to multiple Veras	10
2. Mirroring individual device variables from openLuup to Vera	10
3. Synchronising House Mode	10
The user_data.json file	11
More about the port 3480 web server	13
1. File Access	13
2. index.html Files	13
3. CGI processing	14
Something about openLuup log files	15
More about Scenes	16
1. VeraBridge and Remote Scenes	16
Appendix: Directory Structure and Additional Files	17
1. openLuup directory structure and ancillary files	17
2. /usr/bin/GetNetworkState.sh and /etc/cmh/ui	17
3. code for /usr/bin/GetNetworkState.sh	18
Appendix: Undocumented features of Luup	19
Appendix: Unimplemented features of openLuup	20
1. unimplemented luup API calls	20
2. unimplemented HTTP requests	20

Overview

openLuup is an environment which supports the running of some MiOS (Vera) plugins on generic Unix systems (or, indeed, Windows systems.) Processors such as Raspberry Pi and BeagleBone Black are ideal for running this environment, although it can also run on Apple Mac, Microsoft Windows PCs, anything, in fact, which can run Lua code (most things can - even an Arduino Yún board.) The intention is to offload processing (cpu and memory use) from a running Vera to a remote machine to increase system reliability.

Running on non-specific hardware means that there is no native support for Z-wave, although plugins to handle Z-wave USB sticks may support this. The full range of MySensors (<http://www.mysensors.org/>) Arduino devices are supported though the Ethernet Bridge plugin available on that site. A plugin to provide a bi-directional 'bridge' (monitoring / control, including scenes) to remote MiOS (Vera) systems is provided in the openLuup installation.

openLuup is extremely fast to start (a few seconds before it starts running any created devices startup code) has very low cpu load, and has a very compact memory footprint. Whereas each plugin on a Vera system might take ~4 Mbytes, it's far less than this under openLuup, in fact, the whole system can fit into that sort of space. Since the hardware on which it runs is anyway likely to have much more physical memory than current Vera systems, memory is not really an issue.

There is no built-in user interface, but we have, courtesy of @amg0, the most excellent altUI: Alternate UI to UI7 (see the Vera forum board <http://forum.micasaverde.com/index.php/board,78.0.html>) An automated way of installing and updating the ALTUI environment is now built-in to openLuup. There's actually no requirement for any user interface if all that's needed is an environment to run plugins.

Devices, scenes, rooms and attributes are persisted across restarts. The startup initialisation process supports both the option of starting with a 'factory-reset' system, or any saved image, or continuing seamlessly with the previously saved environment. A separate utility is provided to transfer a complete set of uncompressed device files and icons from any Vera on your network to the openLuup target machine.

For compatibility with Vera systems, openLuup has been written in Lua version 5.1 and requires the appropriate installation to be loaded on your target machine. Installation is not trivial, but it's not hard either. **I would urge you to read this document carefully and then follow the installation steps.**

2016, Release 8.5 (v16.8.5)

Release 8.5 is a comprehensive (but not totally complete) implementation of Luup, and includes a set of features which are generally sufficient to run a number of third-party plugins. The latest additions extend the number of compatible plugins.

The installation section below has been considerably changed (and simplified.)

What openLuup does:

- runs on any Unix machine - I often run it within my development environment on a Mac
- also runs under Windows. However, plugins which spawn commands using the Lua `os.execute()` function may not operate correctly
- runs the ALTUI plugin to give a great UI experience
- runs the MySensors Arduino plugin (ethernet connection to gateway only) which is really the main goal - to have a Vera-like machine built entirely from third-party bits (open source)
- includes a bridge app to link to remote Veras (which can be running UI5 or UI7 and require no additional software.)
- runs many plugins unmodified – particularly those which just create virtual devices (eg. DataYours, iPhoneLocator, Netatmo, ...)
- uses a tiny amount of memory and boots up very quickly (a few seconds)
- supports scenes with timers and ALTUI-style triggers
- has its own port 3480 HTTP server supporting multiple asynchronous client requests
- has a fairly complete implementation of the Luup API and the HTTP requests
- has a simple to understand log structure - written to `LuaUPnP.log` in the current directory - most events generate just one entry each in the log.
- writes variables to a separate log file for ALTUI to display variable and scene changes.

What it doesn't do:

- Some less-used HTML requests are not yet implemented, eg. `lu_invoke`.
- Doesn't support the `<incoming>` or `<timeout>` action tags in service files, but does support the device-level `<incoming>` tag.
- Doesn't directly support local serial I/O hardware (there are work-arounds: `ser2net`)
- Won't run encrypted, or licensed, plugins.
- Doesn't use lots of memory.
- Doesn't use lots of cpu.
- Doesn't constantly reload (like Vera often does, for no apparent reason.)
- Doesn't do UPnP (and never will.)

If you like openLuup, do please consider donating something to [Cancer Research UK](https://www.justgiving.com/DataYours/) at <https://www.justgiving.com/DataYours/>

Installation

NOTE: this has hugely changed from previous versions - with the addition of a new openLuup_install utility and built-in Plugins page.

There are just a few basic installation steps to set up an openLuup system running the ALTUI interface and bridging to a remote Vera:

1. install Lua and some of its packages on your target machine
2. install openLuup itself using the openLuup_install.lua file
3. install and run the VeraBridge plugin from the AltUI Plugins page

That's all that's needed!

1. Lua installation

For compatibility with Vera systems, openLuup has been written in Lua version 5.1 and requires the appropriate installation to be loaded on your target machine. You will also need the LuaSocket library, LuaFileSystem, and, for secure (SSL) network connections, the LuaSec library. It's not difficult, but the guide assumes some familiarity with Linux-type systems (although much is also applicable to Windows.)

It's target machine dependent, but few of the simpler cases are:

RASPBIAN (RASPBERRY PI)

(Lua 5.1 is pre-installed)

```
# sudo apt-get install lua-socket
# sudo apt-get install lua-filessystem
# sudo apt-get install lua-sec
```

DEBIAN (BEAGLEBONE BLACK)

```
# apt-get install lua5.1
# apt-get install lua-socket
# apt-get install lua-filessystem
# apt-get install lua-sec
```

OPEN-WRT (ARDUINO YUN)

(Lua 5.1 is pre-installed)

```
# opkg update
# opkg install luasocket
# opkg install luafilesystem
# opkg install luasec
```

Other machines may require source code compilation of the libraries, but that is beyond the scope of this document.

2. openLuup Installation

This is *very* straight-forward: create a directory for the openLuup installation on your machine (usually **/etc/cmh-ludl/**) and place this file **openLuup_install.lua** there.

```
-- first-time download and install of openLuup files from GitHub

local lua = "lua5.1"      -- change this to "lua" if required

local x = os.execute
local p = print

p "openLuup_install    2016.06.08    @akbooer"

local http  = require "socket.http"
local https = require "ssl.https"
local ltn12 = require "ltn12"
local lfs   = require "lfs"

p "getting latest openLuup version tar file from GitHub..."

local _, code = https.request{
    url = "https://codeload.github.com/akbooer/openLuup/tar.gz/master",
    sink = ltn12.sink.file(io.open("latest.tar.gz", "wb"))
}

assert (code == 200, "GitHub download failed with code " .. code)

p "un-zipping download files..."

x "tar -xf latest.tar.gz"
x "mv openLuup-master/openLuup/ ."
x "rm -r openLuup-master/"

p "getting dkjson.lua..."
_, code = http.request{
    url = "http://dkolf.de/src/dkjson-lua.fsl/raw/dkjson.lua?name=16cbc26080996d9da827df42cb0844a25518eeb3",
    sink = ltn12.sink.file(io.open("dkjson.lua", "wb"))
}

assert (code == 200, "GitHub download failed with code " .. code)

p "creating required files and folders"
lfs.mkdir "files"
lfs.mkdir "icons"

local vfs = require "openLuup.virtualfilesystem"

local function add_vfs_file (name)
    local f = io.open (name, "wb")
    f: write (vfs.read (name))
    f: close ()
end
```

```

add_vfs_file "index.html"

local reload = "openLuup_reload"
local pathSeparator = package.config:sub(1,1)
if pathSeparator ~= '/' then reload = reload .. ".bat" end

add_vfs_file (reload)

p "initialising..."

x "chmod a+x openLuup_reload"

local s= require "openLuup.server"
local ip = s.myIP or "openLuupIP"

p "downloading and installing AltUI..."
x (lua .. " openLuup/init.lua altui")

x "./openLuup_reload &"
p "openLuup downloaded, installed, and running..."
p ("visit http://" .. ip .. ":3480 to start using the system")

-----

```

Run the file using the command line:

```
$ lua5.1 openLuup_install
```

If successful, the script produces console output like this:

```

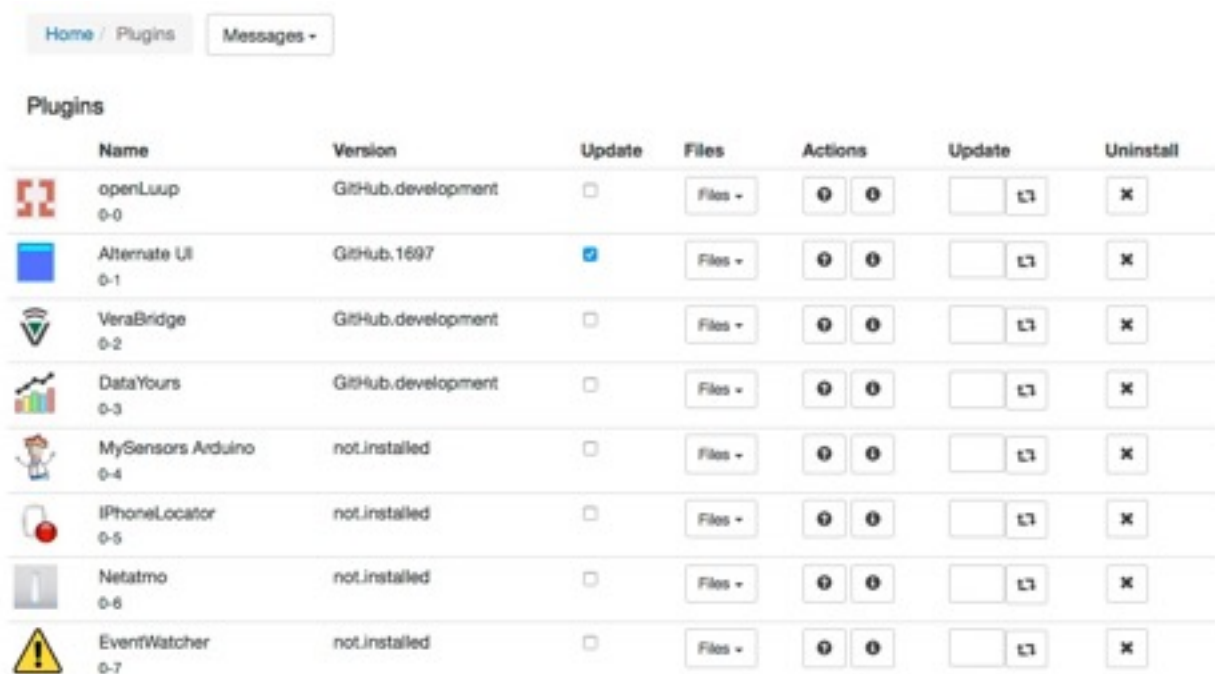
# lua5.1 install.lua
openLuup_install 2016.06.08 @akbooer
getting latest openLuup version tar file from GitHub...
un-zipping download files...
initialising...
downloading and installing AltUI...
Wed Jun 8 18:16:35 2016 device 0 '_system_' requesting reload
openLuup downloaded, installed, and running...
visit http://172.16.42.131:3480 to start using the system
#









































```

and browsing the reported URL will take you to the AltUI interface and show two devices: openLuup and AltUI. From here on, the interface can be used to configure the system.

3. Linking to Vera

Using the AltUI menus to navigate More > Plugins takes you to the plugin page:

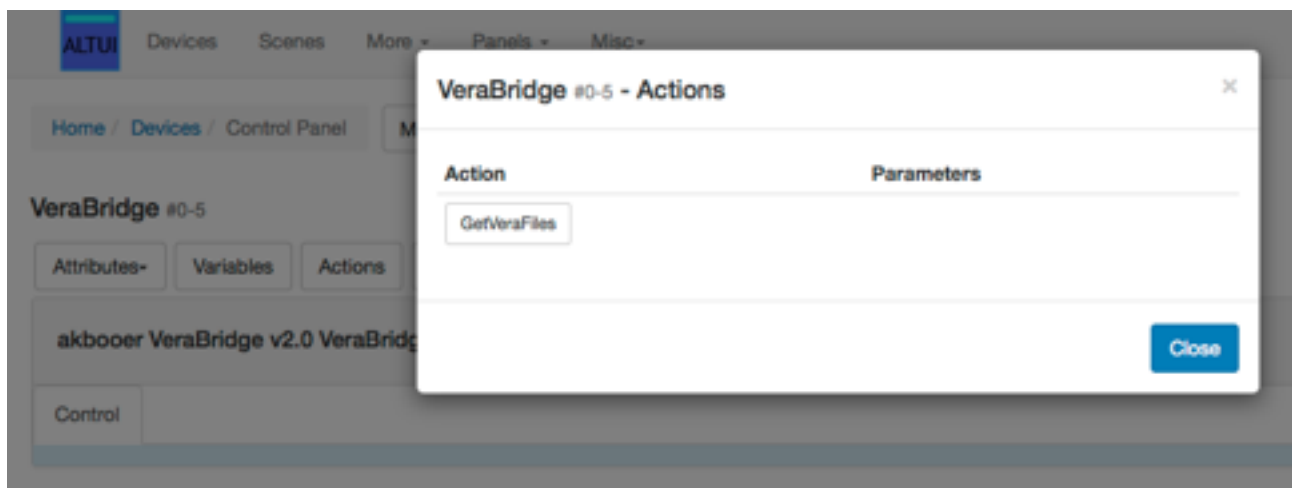


Name	Version	Update	Files	Actions	Update	Uninstall
 openLuup 0-0	GitHub.development	<input type="checkbox"/>	Files -	 	<input type="checkbox"/> 	
 Alternate UI 0-1	GitHub.1697	<input checked="" type="checkbox"/>	Files -	 	<input type="checkbox"/> 	
 VeraBridge 0-2	GitHub.development	<input type="checkbox"/>	Files -	 	<input type="checkbox"/> 	
 DataYours 0-3	GitHub.development	<input type="checkbox"/>	Files -	 	<input type="checkbox"/> 	
 MySensors Arduino 0-4	not installed	<input type="checkbox"/>	Files -	 	<input type="checkbox"/> 	
 iPhoneLocator 0-5	not installed	<input type="checkbox"/>	Files -	 	<input type="checkbox"/> 	
 Netatmo 0-6	not installed	<input type="checkbox"/>	Files -	 	<input type="checkbox"/> 	
 EventWatcher 0-7	not installed	<input type="checkbox"/>	Files -	 	<input type="checkbox"/> 	

Simply clicking on the Update button for VeraBridge will install that plugin.

You have to go to the Devices page and the Attributes tab of the new VeraBridge in order to enter the IP address of your Vera. Moving to the Misc > Reload Luup Engine menu will reload and the bridge should now have connected to that Vera and display its devices and scenes as 'clones' on openLuup.

There is just a little further configuration to do for bridged devices.



None of the device files or icons have yet been downloaded to openLuup, so the icons will be missing and also devices will not have their full functionality. The bridge allows an easy way to retrieve these files: simply go to the **Action** tag of the bridge's control panel and press the **GetVeraFiles** button.

This will copy all device files (D_xxx.xml, D_xxx.json, I_xxx.xml, S_xxx.xml, J_xxx.js, etc...) into **/etc/cmh-ludl/files/** and the icons (xxx.png) in **/etc/cmh-ludl/icons/** and reload the system. After a possibly required browser refresh, the device icons and full functionality should be in place.

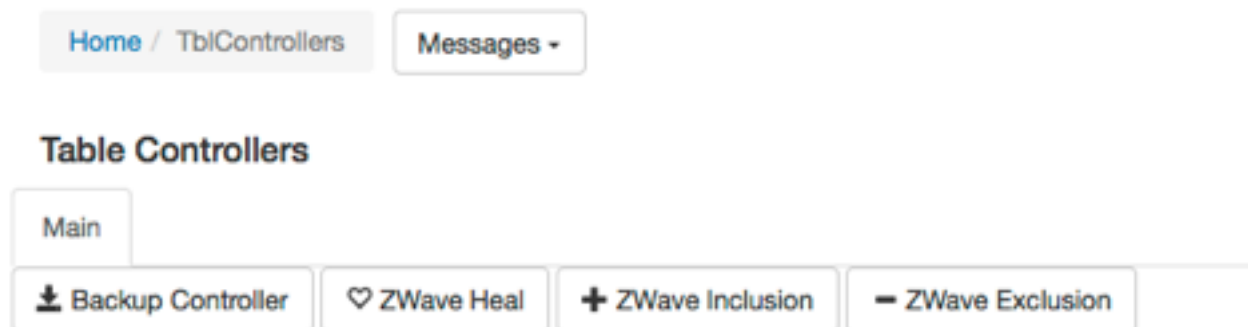
Read more about VeraBridge capabilities in a later section of this guide.

Backing up the system

AltUI makes it very easy to create a backup of your system configuration – all of which is defined in one (large) JSON-encoded file: **/etc/cmh-ludl/user_data.json**.

The More > Controllers page provides a button to save a copy to **/etc/cmh-ludl/backup/** in a file named something like **backup.openLuup-88800000-2016-05-01.lzap**. Repeated backups on the same day overwrite one another. Different days have unique names and are not automatically purged.

This file is a binary compressed version of the user_data.son file, and typically 5 to 6 times smaller. It may be used as a parameter to the openLuup_reload script and the initialisation process will uncompress it automatically.



More about VeraBridge

The VeraBridge plugin is an openLuup plugin which links to remote Veras. Unlike the built-in Vera 'bridge' capability (which, I think, uses UPnP discovery) this has no limitation as to the Vera firmware version that it links to and you can quite happily run multiple copies of this linking to UI5 and UI7 remote machines (which, themselves, require no special software installation.)

VeraBridge provides local clones of devices and scenes from a remote Vera:

- reflects the status of the remote devices (ie. device variables)
- provides, through ALTUI, a display panel and controls
- allows control of the remote devices (eg. on/off, dimming, ...) through the usual luup.call_action mechanism (or the control panel)
- makes remote scenes visible locally, allowing timers and triggers to be added
- maintains a local variable to reflect the remote machine's House Mode
- maintains a local variable showing the time of last contact with remote machine
- capability to lock the openLuup House Mode to the bridged machine or *vice-versa*.
- 'reverse bridging' or 'mirroring' of specified device variables: openLuup variables may be written to devices on the bridged Vera so that plugins and scenes there can be influenced by plugins and scene logic running under openLuup.
- the ability to download all device files and icons from the bridged Vera.

The device numbering is different from that on the remote machine, being incremented by multiples of 10000 for each different bridge, and one of VeraBridge's main functions is to intercept actions on the local devices and pass them to the remote ones. What does NOT work is to set a variable on a local device and expect that variable on the corresponding remote device to change.

Since it is possible (but fairly unlikely) that the VeraBridge will miss an update to a Vera device variable, every so often (actually, about every minute) it scans the whole lot and updates all the cloned devices on openLuup. This means that WHATEVER changes you make to cloned device variables will get overwritten frequently. The motto is this... do NOT write variables to cloned devices.

Running under openLuup, VeraBridge supports *any* serviceId/action command request (ie. those *not* returning device status parameters) This covers most device control scenarios.

The bridge device also has a variable which reflects the remote House Mode (if the remote system is UI7 or greater.) Additionally, the bridge can 'mirror' the house mode in either direction, synchronising the mode of both (or more) machines.

1. Bridging to multiple Veras

Multiple bridges may be installed (through the Create button on the devices page.)

The first Vera linked to is special, for a number of reasons:

TBD...

2. Mirroring individual device variables from openLuup to Vera

Following (sort-of) @logread's original proposal, and now that Lua Startup can be edited in AltUI, you can write, in Startup, something like:

```
-- Mirrors syntax:
-- <VeraIP>
-- <Vera deviceId> = <openLuup deviceId>.<serviceId of the variable>.<variable
name>
-- ...

luup.attr_set ("openLuup.mirrors", [[
192.168.99.99
82 = 15.urn:upnp-org:serviceId:TemperatureSensor1.CurrentTemperature
83 = 16.urn:micasaverse-com:serviceId:HumiditySensor1.CurrentLevel
85 = 17.urn:cd-jackson-com:serviceId:SystemMonitor.memoryAvailable
85 = 17.urn:cd-jackson-com:serviceId:SystemMonitor.systemLuupRestartTime
]])
```

Which can be extended to include other Vera bridges.

3. Synchronising House Mode

The bridge has a device variable HouseModeMirror, which you can set to one of three values:

- "0" : no mirroring,
- "1" : local mirrors remote machine,
- "2" : remote mirrors local machine.

You have to reload to make this change effective. Because of Vera's built-in delay in changing HouseMode, mode 2 can take a while (typically 30 seconds or more) to kick in, but they should synchronise in the end.

Note that it's not easily possible to have this synchronisation work in both directions simultaneously because of a race condition that arises from the latency of Vera changing modes. You can, however, have openLuup mirror one bridged Vera's mode and have other bridged Veras which mirror that.

The user_data.json file

Configuration changes to the system happen in different ways in the three major phases of normal running, startup and shutdown:

STARTUP

- the default behaviour is to look for a JSON file called `user_data.json` in the current directory (`/etc/cmh-ludl/`) loading the device/room/scene configuration from that.
- an optional startup parameter is one of:
 - the word `reset`, which forces a ‘factory reset’, or
 - a `user_data` JSON formatted filename, or
 - a Lua filename to be run in the context of a factory-reset system (with no rooms or scenes, and only devices 1 (Gateway) and 2 (Z-wave controller) defined)

RUNNING

- every 6 minutes the system configuration is check-pointed to the file `user_data.json`, regardless of the name or type of the startup file. This will capture device variable and attribute changes, scene creation/deletions, etc.
- logged events are written to the log files as they happen and not cached.

SHUTDOWN

- on `luup.reload`, or full exit, the configuration will be written to the file `user_data.json` in the current directory
- on `luup.reload`, or any other configuration change requiring a reload (eg. new child devices created) the process will exit with status 42
- on exit (from the HTTP request `id=exit`) will exit with status of 0 (successful exit)

This means that it is easy to save and restore arbitrary configurations, or start from scratch. Since any reload will cause the process to exit, it's necessary to launch `openLuup` from within a script, in order to restart automatically. A Unix shell script to do this, `openLuup_reload`, is included in the distribution (in the `openLuup - Utilities` directory) and shown here. For Windows, see the relevant Appendix.

```
#!/bin/sh
#
# reload loop for openLuup
# @akbooer, Aug 2015
# you may need to change 'lua' to 'lua5.1' depending on your install

lua openLuup/init.lua $1

while [ $? -eq 42 ]
do
    lua openLuup/init.lua
done
```

An example Lua startup code, shipped with the `openLuup` installation and listed in an Appendix, shows how to make a minimal system running ALTUI, Arduino Gateway, and a VeraBridge to a remote Vera. The **bold lines** show where you might change the customisation.

Whatever else you do, backup your functional user_data.json file.

You can always revert to this if you get stuck, using it as the input parameter to openLuup_reload. So long as it is call something different, it will NOT be over-written, so this is a very easy way to switch between different configurations.

One other ‘safety net’ that openLuup gives you in terms of restoring a configuration is to use a startup.lua file (again, you can have many of these called different things.) The Lua syntax of the file gives a very simple and readable description of the configuration and, as the example shows, it’s very easy to create devices, rooms, etc. Scenes are more difficult, however. In fact, at the moment, you can’t – you’d have to recreate them manually through the ALTUI interface.

If you DO recreate a system from a startup.lua file, then be aware that every plugin that creates new child devices will, in turn, request a reload. Depending on what the plugin does in its own startup code (access REST APIs over the network, local files, delayed startups...) this can take quite a long time and you should see the system restart messages. During this time the ALTUI interface (if you have configured it) will not be accessible.

Even after a system reboot, the only thing required to get openLuup restored to its previous state is to start the openLuup_reload script (with no parameters) since this just picks up the latest user_data.json file and runs with it.

More about the port 3480 web server

As well as supporting Luup HTML requests, the openLuup server on port 3480 has some of the functionality of any normal server, but also some special features:

1. File Access

The root directory for the port 3480 server is `/etc/cmh-ludl/` (or wherever the openLuup process is started.) However, in addition, the server will search in `/etc/cmh-lu/` (actually `../cmh-lu/`) if it fails to find the requested file. This mimics Vera's use of that directory. It means that you can put `.xml`, `.json`, and other files there for convenience.

Note, however, that a plugin which expects to find a file in `/etc/cmh-ludl/` using `io.open()` will NOT automatically find it if it resides in `/etc/cmh-lu/`.

The Lua package path has also been modified to search `/etc/cmh-lu/` so that required Lua modules may also be located there. Note also that any files here are not in compressed for (.lzo) as they are in Vera.

Additionally, file-based icon references in `.json` files are redirected during loading to access the `/etc/cmh-ludl/icons/` directory through the port 3480 server. This means that the requests do not go from AltUI to port 80, but to port 3480, which in turn means that device icons work over a secure connection to port 3480 only. It also means that the icon directories under `/www/cmh/skins/default/...` are not required. Existing HTTP icon references are untouched.

2. index.html Files

The server looks for an `index.html` file if the request is for a directory (ends with `/`). This means that an `index.html` file may be used to present any web page you like, but in particular it may be used to redirect the request. For example:

With the following text in `/etc/cmh-ludl/index.html`

```
<!DOCTYPE html>
<html>
  <head>
    <!-- HTML meta refresh URL redirection -->
    <meta http-equiv="refresh"
      content="0; url=/data_request?id=lr_ALTUI_Handler&command=home#">
  </head>
</html>
```

then `http://openLuupIP:3480/` redirects to AltUI.

Once again, this is useful functionality for access over a secure connection.

3. CGI processing

The openLuup HTTP server has a WSAPI connector for openLuup to allow it to run WSAPI (Lua) applications as CGIs.

This means that arbitrary web server functionality can be added to openLuup simply by writing a short, stand-alone, Lua module. The first time a CGI URL is accessed, the module is compiled and loaded. Thereafter, it runs just as quickly as any built-in server code. The implementation adheres exactly to the (brief) documentation here: <http://keplerproject.github.io/wsapi/manual.html>.

The connector supports CGI code in the `cgi/`, `cgi-bin/`, and `upnp/` directories in `/etc/cmh-ludl/`. So, for example, taking *exactly* the code from the simple example in the above documentation:

```
#!/usr/bin/env wsapi.cgi

module(..., package.seeall)

function run(wsapi_env)
  local headers = { ["Content-type"] = "text/html" }

  local function hello_text()
    coroutine.yield("<html><body>")
    coroutine.yield("<p>Hello Wsapi!</p>")
    coroutine.yield("<p>PATH_INFO: " .. wsapi_env.PATH_INFO .. "</p>")
    coroutine.yield("<p>SCRIPT_NAME: " .. wsapi_env.SCRIPT_NAME .. "</p>")
    coroutine.yield("</body></html>")
  end

  return 200, headers, coroutine.wrap(hello_text)
end
```

...and putting it into `/etc/cmh-ludl/cgi/hello.lua`, allows you to invoke it with

```
http://openLuupIP:3480/cgi/hello.lua
```

...and get the returned page:

Hello Wsapi!

PATH_INFO: /

SCRIPT_NAME: /cgi/hello.lua

Any other CGI action can be implemented in the same way.

Note that shell scripts and other CGI implementations are not yet supported by the port 3480 server, but this mechanism does enable some of the key CGIs used by Vera to be emulated through a small amount of effort.

Something about openLuup log files

openLuup write its main log file to `/etc/cmh-ludl/LuaUPnP.log`. To keep file sizes manageable, it rotates the log from time to time (after about 2,000 lines, which makes the file size about 250 Kb.) The five most recent log files are versioned with the suffix `.1 / .2 / ...etc.`, so you can go back and see what happened just before a reload. The reason for the reload should be one of the last lines in the log. Unlike Vera, this doesn't just happen randomly, but in response to either a user request or a device doing something which requires a reload (adding a new child device, for example.)

openLuup also maintains another log, which is a subset of the main one, and contains only device variable changes and scene invocations. Those entries are written to a log file, if the `/var/cmh/` directory exists, in `/var/cmh/LuaUPnP.log`, with colour-highlighted variable names, as in Vera's log. The OS command that comes as standard in AltUI tails this log, and it's an easy way just to see the major changes happening in the system, without being cluttered by other activities which write to the log file. IT is also used by AltUI to provide a history of device variable changes. This log file is not versioned, and reset after 5000 lines, so it is possible to come to it sometimes and find it almost empty. This file is also essential for the variable history function of AltUI to operate – the file is scanned for the relevant variable changes which are presented as the change history under that button of the device variable.

openLuup doesn't, internally, use the same log level numbering system as Vera, trying to keep things simple. Log entries created by devices always carry their device ID. Here's a log entry made by device #7

```
luup_log:7: Arduino: urn:micasaverde-com:serviceId:HaDevice1,LastUpdateHR, 09:42, 42
```

If you're trying to troubleshoot a single plugin, then I always recommend that you do it in a system which only has that plugin. That way, almost all of the log entries are relevant. Of course, if it's complex multi-device logic, then you can't do that.

A system log from a system after a 'factory reset' initialisation is shown below:

```
2016-06-09 09:09:28.783   :: openLuup STARTUP ::
2016-06-09 09:09:28.784   openLuup.init::          version 2016.06.06  @akbooper
2016-06-09 09:09:28.796   openLuup.scheduler::     version 2016.04.30  @akbooper
2016-06-09 09:09:28.797   openLuup.wsapi::         version 2016.05.30  @akbooper
2016-06-09 09:09:28.797   openLuup.server::        version 2016.06.01  @akbooper
2016-06-09 09:09:28.813   openLuup.github::        version 2016.05.30  @akbooper
2016-06-09 09:09:28.813   openLuup.plugins::       version 2016.06.08  @akbooper
2016-06-09 09:09:28.815   openLuup.scenes::        version 2016.05.19  @akbooper
2016-06-09 09:09:28.817   openLuup.rooms::         version 2016.04.30  @akbooper
2016-06-09 09:09:28.818   openLuup.chdev::         version 2016.06.02  @akbooper
2016-06-09 09:09:28.818   openLuup.userdata::      version 2016.06.08  @akbooper
2016-06-09 09:09:28.819   openLuup.io::            version 2016.04.30  @akbooper
2016-06-09 09:09:28.819   openLuup.luup::          version 2016.06.06  @akbooper
2016-06-09 09:09:28.821   openLuup.requests::      version 2016.06.04  @akbooper
2016-06-09 09:09:28.836   luup.create_device:: [1] D_ZWaveNetwork.xml /
I_ZWave.xml /
2016-06-09 09:09:28.841   luup.create_device:: [2] D_openLuup.xml /
I_openLuup.xml / D_openLuup.json
2016-06-09 09:09:28.841   openLuup.init:: loading configuration reset
2016-06-09 09:09:28.841   openLuup.luup:0: device 0 '_system_' requesting reload
2016-06-09 09:09:28.841   luup.reload:0: saving user_data
2016-06-09 09:09:28.845   openLuup.luup:0: exiting with code 42 - after 0.0
hours
```

More about Scenes

Scene timers and actions can be defined with the AltUI interface in exactly the same way as on Vera. Triggers, however, are treated differently, because the Vera approach, itself based on UPnP definitions in various files, is deeply flawed and inflexible. Instead, the AltUI-supported mechanism of device variable triggers are used.

Using the appropriate button under the scene / trigger definition menu, you may select any device and variable as a trigger. The trigger will fire whenever the variable changes value, and using a small amount of Lua code, or the blocky graphical interface, you can construct arbitrary logic expressions combining, times, old and new variable values, other device variables, in fact almost anything you want, to determine whether or not the scene should actually execute.

1. VeraBridge and Remote Scenes

A scene that has been created by VeraBridge is simply a perfectly normal openLuup scene (hence totally isolated from any scene code on your actual Vera) with one-line in the Lua code section (which you're able to view and edit) which fires off the remote scene.

This means you can add local triggers, timers, even other Lua code (before the the statement that runs the remote scene!) without any code impact on your actual Vera... very comforting when developing!

So, really, these types of scenes are not clones of the Vera scenes, but merely ways to trigger them remotely.

No aspect of the remote scenes can be changed – it is not an editor for remote scenes. You can't edit scenes from a remote Vera on openLuup if they are bridged by the VeraBridge plugin – you have to edit them on the original Vera.

They are really there as a basis for re-writing them to operate entirely locally in the openLuup environment as you migrate plugins and scene logic from Vera to openLuup. The bridge's capability to mirror openLuup variables to actual device variables on Vera makes this transition easier in some cases.

Appendix: Directory Structure and Additional Files

The installations steps 1, 2, 3, give you a functioning system, linked to a remote Vera. But to allow as many plugins as possible to run on openLuup, you may need additional files and directories.

1. openLuup directory structure and ancillary files

Some of this structure is already created by the install process, other parts are totally optional, some are required by particular plugins.

- /etc
 - /cmh
 - ui
 - /cmh-lu
 - /cmh-ludl
- /usr
 - /bin
 - GetNetworkState.sh

2. /usr/bin/GetNetworkState.sh and /etc/cmh/ui

Some plugins (eg. Sonos) use a shell command `/etc/bin/GetNetworkState.sh` to determine the machine's IP address. I used to provide this (actually for ALTUI, but it doesn't need it now.) So you need to create that file with the contents shown in the Appendix.

On a Windows machine you may need to use the simpler hard-coded address version of the script shown there. I'm not actually sure that this works.

At least one plugin (IOSPush) requires the file `/etc/cmh/ui` with the single line (the digit seven):
7 Appendix: openLuup_reload.bat for Windows

For Windows, you will need an alternative shell file to run the basin openLuup reload loop. I'm indebted to @vosmont for the following information which I've copied directly from the post here <http://forum.micasaverde.com/index.php/topic,34480.msg259829.html#msg259829>

For the moment, I use openLuup on Windows.

Install <http://luadist.org/> (contains Lua 5.1 and all the needed libraries)

Copy this file "openLuup_reload.bat" in "openLuup\etc\cmh-ludl", and change "LUA_DEV" according to LuaDist folder.

```
@ECHO OFF
SETLOCAL
SET LUA_DEV=D:\devhome\app\LuaDist\bin
SET CURRENT_PATH=%~dp0
ECHO Start openLuup from "%CURRENT_PATH%"
ECHO.
```

```

CD %CURRENT_PATH%
"%LUA_DEV%\lua" openLuup\init.lua %1

:loop
IF NOT %ERRORLEVEL% == 42 GOTO exit
"%LUA_DEV%\lua" openLuup\init.lua
GOTO loop

:exit

```

3. code for /usr/bin/GetNetworkState.sh

Some plugins (eg. Sonos, DLNA, Squeezebox, ...) require an external shell script (part of a standard Vera installation) to define the host machine IP address.

There's two basic ways to implement this.

This file can either be VERY simple:

```
echo -n 172.16.42.88
```

...with the IP address appropriately set (manually)

...or a more sophisticated approach using a Lua script to return the result (automatic) which is described here:

<http://forums.coronalabs.com/topic/21105-found-undocumented-way-to-get-your-devices-ip-address-from-lua-socket/>

```

#!/usr/bin/env lua
-----
-- discover main IP address of machine and write to standard output
-- http://forums.coronalabs.com/topic/21105-found-undocumented-way-to-get-your-
devices-ip-address-from-lua-socket/
-----
local socket = require "socket"
function myIP ()
    local mySocket = socket.udp ()
    mySocket:setpeername ("42.42.42.42", "424242") -- arbitrary IP/PORT
    local ip = mySocket:getsockname ()
    mySocket:close()
    return ip or "127.0.0.1"
end
io.write (myIP())
-----

```

Appendix: Undocumented features of Luup

The documentation for Luup is poor: out of date, misleading, incomplete, unclear, and sometimes just plain wrong. During the implementation of openLuup a number of undocumented 'features' have come to light. In some cases, these features are used by various plugins, either deliberately or unknowingly, through sins of commission or omission. As a result, I've had to include them in the openLuup implementation.

If you're a developer, please try NOT to rely on these things in your plugin code.

Here's what I've found - if you know more, let me know.

- **lul_device** – this variable is often included in callback function parameter lists to indicate the target device, and that's fine. However, it ALSO turns out to be in scope in the whole body of a plugin's Lua code. Some plugins rely (possibly inadvertently) on this feature.
- **nil device parameter in luup.variable_watch** – whilst the use of a nil variable parameter to watch ALL service variables is documented, the use of a nil DEVICE is not, but works as expected: the callback occurs a change in ANY device with an update to the given serviceld and variable (thanks for @vosmont for that information.)
- **nil device parameter in luup.variable_set/get** – it appears that when called from device code, a missing device parameter gets the current luup.device variable value substituted.
- **urn:micasaverde-com:serviceld:HaDevice1, HideDeleteButton** – according to @reneboer: *"If you set that to 1 on UI7 it will not show the delete button at the bottom of the device Control Panel. I now use this to hide that on the child devices that are under full control of the parent device, including proper deletion."*
- **the ordering of the <files> tag and the <functions> tag matters** – in implementation files, they are concatenated in that order and it matters to the scope of local variables defined there (thanks to @logread and @cybrimage for that nugget.)

Appendix: Unimplemented features of openLuup

openLuup is, as of this time, an unfinished work. The following features are known to be unimplemented, poorly implemented, or non-functional at this time, for a variety of reasons.

1. unimplemented luup API calls

- **luup.devices_by_service** – the definition of the functionality here [Luup Lua Extensions](#) is not adequate to make an implementation.
- **luup.job.*** – all functions in this module (status, set, setting) are empty stubs.
- **luup.job_watch** – not yet implemented.
- **luup.require** – undocumented.
- **luup.xj** – undocumented.

2. unimplemented HTTP requests

- **id=scene** – only create, delete, list, and rename are implemented (ie. no interactive creation of scenes.) Also note that scene triggers (or notifications) are not stored.
- **id=action** – the virtual category 999 is not implemented. Actions on groups of devices defined by category is not implemented.
- **id=finddevice** – not implemented.
- **id=resync** – not implemented.
- **id=archive_video** – not implemented.
- **id=jobstatus** – not implemented.
- **id=invoke** – not implemented.
- **id=relay** – not implemented.