python-brisa Documentation

Release 0.10.0

Andre Dieb Martins
Leandro Melo de Sales
Felipe Coutinho

June 13, 2009

CONTENTS

Contents:

CHAPTER

ONE

ABOUT

This document describes the BRisa Project, its goals and what it provides.

1.1 BRisa Project

Under development since 2007 at the Laboratory of Embedded Systems and Pervasive computing located at Universidade Federal de Campina Grande - UFCG (Campina Grande - Brazil), the project general goal is to provide UPnP solutions.

The first of these UPnP solutions was python-brisa, a UPnP framework written in *Python* for deploying devices and control points. News and updates will always be posted in our front page.

1.2 People

Team Leader and Project Manager: Leandro de Melo Sales <*leandroal at gmail dot com*>

Student Developers (python-brisa):

- André Dieb Martins - *andre.dieb at gmail dot com*
- Arthur de Souza Ribeiro - *arthurdesribeiro@gmail.com*
- Felipe Coutinho - *felipelcoutinho at gmail dot com*

1.3 Releases

Release schedule will always be posted in our front page.

CHAPTER

TWO

GLOSSARY

This is a basic glossary recommended reading before starting reading the documentation itself.

BRisa Project project under development at Laboratory of Embedded Systems and Pervasive Computing, financed by INdT.

python-brisa a python upnp framework developed by the BRisa Project.

brisa-applications or brisa-apps applications developed by the BRisa Project that uses python-brisa.

upnp also known as UPnP, short for universal plug and play, protocol defined by the UPnP Forum. Documents can be found at http://www.upnp.org.

device for our purposes, when refering to a device, we will be refering to a UPnP device, that is, an entity capable of joining the UPnP network (thus capable of performing its role on the UPnP Architecture.

service for our purposes, a service should be considered an UPnP service, that is, a service provider of actions. For example, a Content Directory service provides a Browse action that enables others to browse through a remote directory on the service location.

control point an entity of the UPnP network capable of controlling devices

reactor the gears or engine that powers the framework. This can be simply put analogous as a car engine - you can replace it by another one that provides the basic functionalities your car requires. python-brisa provides some engine options integrated with different frameworks, such as Gtk, Gobject, Qt, and so on.

CHAPTER

THREE

CORE

These are the core modules of *python-brisa*. Each section describes a module and how to use it.

3.1 `reactor` — Reactor facilities

3.1.1 Choosing a Reactor

The first thing that may be done on a python-brisa application is to setup the framework gears, that is, choose a reactor.

The choice of a reactor is very simple and depends mainly on your application. If your application requires somehow Gtk, Qt, ecore or even gobject, you should choose the reactor that attends that demand. Besides having a **default** python-pure reactor, *python-brisa* implements:

- glib/gobject
- gtk2
- ecore
- Qt

If your application do not demand any of these libraries/frameworks, then you should probably use the default reactor. The next section explains how to install the default reactor and subsequent sections explains how to use the reactor.

3.1.2 Default Reactor

Installing the default reactor:

```
from brisa.core.reactors import install_default_reactor
reactor = install_default_reactor()
```

Once installed, the reactor cannot be replaced during the same runtime.

3.1.3 Retrieving a Installed Reactor

Once installed, the reactor can be retrieved the same way it was installed (by calling install_default_reactor() or e.g. Gtk2Reactor()). The reactor will be installed on module *brisa.core.reactor* and can also be retrieved with:

```
from brisa.core import reactor
```

3.1.4 Gtk2Reactor Example

For instance, if your application contains a Gtk GUI, you must install the Gtk2Reactor, action illustrated below:

```
from brisa.core.reactors import Gtk2Reactor
reactor = Gtk2Reactor()
```

The most important thing here is to notice that once installed, this reactor will use Gtk internally, so your old calls to `gtk.main()`, `gtk.main_quit()` will be equal to `reactor.main()`, `reactor.main_quit()`, respectivelly.

For the sake of organization, we recommend you to use the reactor interface (e.g. use `reactor.main()` instead of `gtk.main()`). It's possible the reactor doesn't work properly if you don't use the interface.

3.1.5 QtReactor Example

If your application contains a Qt GUI or a Qt Core, you must install the QtReactor, action illustrated below:

```
from brisa.core.reactors.qtreactor import QtReactor
reactor = QtReactor()
```

This reactor will use Qt internally, so remember to call `reactor.main()` and `reactor.main_quit()` as we did with Gtk2Reactor. Take a look at **code examples** to see a Qt device and a Qt control point example.

The concepts illustrated above are the same for the other reactors.

3.1.6 Advanced Reactor Usage

The reactor interface defines that reactors must support file descriptors event handling and timers.

File descriptors event handlers and timers can be added before starting the reactor (e.g. `reactor.main()`) and during runtime. Though, they will not work if the reactor hasn't been started.

3.1.7 Creating a Timer

For creating a timer, use the reactor's method `add_timer()`. By providing the interval to wait between calls to the function, you will receive an unique id for the timer. Use it later for removing it from the reactor:

```
from brisa.core import reactor

# Adds a timer
timer_id = reactor.add_timer(interval, my_function)
(...)

# Removes the timer given it's id
reactor.rem_timer(timer_id)
(...)
```

3.1.8 Creating an Event Handler

For creating an event handler, use the reactor's method `add_fd()`. By providing a fd file object, a callback for the event and a type for watch on the event, you will get an unique handler. Use this handler later for removing the event handler from the reactor with the `rem_fd()` method.

Whenever an event of the type specified (read, write or exception) happens on the file, your callback will be called with the following format: `callback(file, event_type)`.

The available *type* flags are:

- EVENT_TYPE_READ
- EVENT_TYPE_WRITE
- EVENT_TYPE_EXCEPTION

Multiple events can be watched by passing an or combination of event types on the *type* parameter (e.g. `type=EVENT_TYPE_READ|EVENT_TYPE_WRITE` for watching both read and write events):

```
from brisa.core import reactor
from brisa.core.ireactor import EVENT_TYPE_READ

def on_ready_to_read(fd, type):
    print 'I read: ', fd.read()

# Add my fd
fd_handler = reactor.add_fd(myfd, on_ready_to_read, EVENT_TYPE_READ);

(...)

reactor.rem_fd(fd_handler)
```

3.2 `config` — Configuration facilities

python-brisa provides a very simple configuration API and a built-in configuration manager.

3.2.1 Using the Built-In Configuration Manager

```
from brisa.core.config import manager

# Retrieving the value of some_section.some_parameter_name
print manager.get_parameter('some_section', 'some_parameter_name')

# Setting some_section.some_parameter_name = 1
manager.set_parameter('some_section', 'some_parameter_name', 1)
```

The default manager contains a default section for framework settings. It is called *brisa* and you can play with the manager's methods on it (e.g. `manager.contains()`, `manager.items()`, and so on).

3.2.2 Direct Access Option

When activated, the direct access option makes the manager perform changes (e.g. `manager.set_parameter()`) directly on the configuration static storage. If not activated, all changes made during runtime will be lost if you don't explicitly call `manager.save()`.

Direct Access can be activated with `manager.set_direct_access(True)` and deactivated the same way, passing False as argument.

3.2.3 Configuration Tool

There's a command-line configuration tool called `brisa-conf`. It can list sections, add/remove sections, set parameters, clean up configuration.

For help on how to use it, try `brisa-conf --help`. It will give you detailed information on how to use it.

We also have a few examples illustrated below.

Listing all sections:

```
brisa-conf -l
```

Listing items of section `brisa`:

```
brisa-conf -i brisa
```

Setting logging output to stdout:

```
brisa-conf -s brisa -p logging_output stdout
```

Setting logging level to DEBUG:

```
brisa-conf -s brisa -p logging DEBUG
```

3.3 `log` — Logging facilities

python-brisa provides a logging module with a colored logging feature.

The logging module is configured by the default manager described on `config` under the default *brisa* section.

The two main parameters that can be adjusted are:

- brisa.logging, determines the highest logging level (ERROR, INFO, CRITICAL, DEBUG, WARNING)
- brisa.logging_output, which determines where the log should appear (commonly used values are *stdout*, *file* - located at ~/brisa.log).

These parameters can be modified through the config module or using the `brisa-conf` tool. Try `brisa-conf -h` for usage.

3.3.1 Global Logger

This module provides a global (or root) logger, which can be used simply by:

```
from brisa.core import log

log.debug('My debug message.')
log.error('My error message.')
log.info('My info message.')
```

3.3.2 Loggers

For retrieving loggers other than the global/root one, use the `log.getLogger()` function as follows:

```
from brisa.core import log

mylogger = log.getLogger('my_logger')
```

3.4 `threaded_call` — Threaded function calls

python-brisa provides a module with capabilities of running function calls on separate threads.

3.5 `network` — Network facilities

python-brisa provides a network module with some functions which will simplify networking related tasks.

3.5.1 Using the parse_url() function

This little example shows a simple and generic use of the parse_url() function

```
from brisa.core.network import parse_url
url = 'http://brisa.garage.maemo.org'
parse_url(url)
```

It will return a 6-tuple: (scheme, netloc, path, params, query, fragment) which in the example above will look like this

```
('http', brisa.garage.maemo.org', '/', '', '', '')
```

3.5.2 Getting the ip of interfaces

When you need to get the ip of one of your interfaces you can do as following

```
from brisa.core.network import get_ip_address, get_active_ifaces
for interfaces in get_active_ifaces():
    get_ip_address(interfaces)
```

This will output strings containing the active interfaces' IP address

3.6 `network_listeners` — Network listening facilities

python-brisa provides a network listening module which will make it easy to implement, for example, a udp server.

3.6.1 Implementing a simple udp server.

Look how simple it's to implement a udp server

```
from brisa.core.reactors import install_default_reactor
reactor = install_default_reactor()

from brisa.core.network_listeners import UDPListeners

def forward_data(data, addr=''):
    # This is the data handler function, in this example it will simply
    # print the received data and the address from where it came
    print data, 'from ', addr

server = UDPListener('239.255.255.250',
                     port,
                     'interface',
                     data_callback = foward_data)

# Be sure to register the UDPListener's stop function before using the
# start method, or the program doesn't end.
server.start()
reactor.add_after_stop_func(server.stop)
reactor.main()
```

This is all it takes to implement a simple udp server

3.7 `webserver` — Webserver facilities

python-brisa provides a WSGI application-side to which any WSGI server can be plugged in.

Along with the WSGI app-side, we provide three adapters for known python webservers:

- **CherrypyAdapter** - Cherrypy
- **CircuitsWebAdapter** - Circuits.web
- **PasteAdapter** - Python-paste

In the simplest case, all you need is to have **at least one** of them installed, and use the webserver without any worry about WSGI, adapters or anything else. See the illustrative code below and the example of the next section:

```
(...)
from brisa.core.webserver import WebServer

webserver = WebServer('MyServerName', 'localhost', 1234)

# Add files, resources
(...)

webserver.start()
```

If you have more than one installed and you want to explicitly use one of them, there are two ways:

1. Set a configuration entry:

```
$ brisa-conf -s brisa -p webserver_adapter X
```

where X is the adapter name (`cherrypy`, `circuits` or `circuits.web`).

2. Choose during runtime:

```
from brisa.core.webserver import WebServer, CircuitsWebAdapter

webserver = WebServer(adapter=CircuitsWebAdapter)
```

For performance reasons we recommend the CircuitsWeb adapter.

3.7.1 Creating a simple webserver

This example shows the webserver usage, along with files and resources. You can find this source **here**.

In a first moment, we setup the reactor, the webserver and add a file to the webserver on the relative path `/hello_world`:

```
from brisa.core.reactors import install_default_reactor
reactor = install_default_reactor()

import os

from brisa.core.webserver import WebServer, StaticFile, CustomResource

# Setup webserver
webserver = WebServer()

# Write and serve a sample file
f = open('/tmp/hello_world', 'w')
f.write('Hello World!')
f.close()

webserver.add_static_file('hello_world', StaticFile('/tmp/hello_world'))
```

The resource class can be used for more complex requests. We create here a Greeter which will greet someone given its name. It is added to the webserver on the relative path `/Greet`:

```
# Serving a resource
class Greeter(CustomResource):

    def get_render(self, uri, params):
        return self

    def say_hello(self, name):
        return 'Hello %s!' % name

    def render(self, uri, request, response):
        params = cherrypy.request.params

        if 'name' in params:
            # http://addr:port/Greet?name=Someone
            return self.say_hello(params['name'])
        else:
```

```
            return 'Hello!'

webserver.add_resource('Greet', Greeter())
```

In the final step, we start the webserver and print the test URLs that can be used to verify the functionality:

```
# Starting the webserver
webserver.start()

print 'Webserver listening on', webserver.get_listen_url()
print 'File URL: %s/hello_world' % webserver.get_listen_url()
print 'Res URL: %s/Greet' % webserver.get_listen_url()
print 'Res test URL: %s/Greet?name=you' % webserver.get_listen_url()

# Block so that the program doesn't quit

reactor.add_after_stop_func(webserver.stop)
reactor.main()

os.remove('/tmp/hello_world')
```

CHAPTER

FOUR

UPNP

These are the UPnP modules of *python-brisa*. Each section describes a module and its usage.

4.1 `base_device` — Basic device classes

The base_device module contains two classes that have all the attributes and methods that are common to all devices.

4.1.1 BaseDevice class

BaseDevice is the base class for devices. On *python-brisa*, devices are represented by objects. For the control point API, a device has a set of methods for dealing with events, variables and so on. For the device API (deploying devices), a device contains methods directed to the deployment.

The BaseDevice class contains attributes that are common for both device-side and control-point-side devices. Even so, it's very unlikely for users to use BaseDevice directly.

Attributes

The BaseDevice class attributes describes the device in some aspects, such as:

- The type of the device
- Its location, uuid and friendly name
- If it has a parent device (embedded device), its manufacturer information
- Informations about the model of the device and its serial number

Methods

The BaseDevice class methods make it possible to deal with the device and its services. They are:

`rem_service_by_id`(*id*)
: It removes the service that matches the given id.

`add_device`(*device*)
: Adds the given device into this device.

`rem_device`(*device*)
: Removes the given device from this device.

add_service(*service*)
Adds the given service into this device.

get_service_by_type(*service_type*)
Returns a service given its type.

rem_service(*service*)
Removes a service, if present on the device.

is_root_device()
Returns True if this device is a root device (it contains embedded devices).

4.1.2 BaseDeviceIcon class

Also a base class for devices, it contains attributes that describes the Device's icon, such as width, height, depth, url and mimetype.

Attributes

The BaseDeviceIcon attributes are:

- Mimetype - mimetype for the icon
- Width - icon width
- Height - icon height
- Depth - icon depth
- Url - the icon's url

Methods

BaseDeviceIcon's methods are:

get_mimetype()
Returns the icon's mimetype in string form.

get_width()
Returns the icon's width in string form.

get_depth()
Returns the icon's depth in string form.

get_url()
Return the icon's url in string form.

4.2 base_service — Basic service classes

BaseService contains all the attributes that are common to services on both control-point and device sides.

4.2.1 Module's methods

format_rel_url(*url*)
Formats the given url from the form path/to to /path/to, if required and returns it as a string.

parse_base_url (*url*)
Parses the given url and return a string in the form scheme://netloc

4.2.2 BaseService class

BaseService is the base class for services. The BaseService class contains attributes that are common for both device-side and control-point-side services. Even so, its very unlikely for users to use BaseService directly.

Attributes

The BaseService's attributes describe some of the service's basic information.

- Service's id.
- Service type.
- Several urls such as scpd url, control url, event and presentation.
- Eventing variables.

4.2.3 BaseStateVariable class

This class represents the service state variables.

Attributes

These attributes describes the state variables of a service which holds them. They are:

- parent_service - Service which holds this variable.
- name - Variable name.
- send_events - Send events option.
- data_type - Data type.
- allowed_values - Values that are allowed.

4.3 ssdp — SSDP Server implementation

python-brisa provides a SSDP (Simple Service Discovery Protocol) module which can be used on the control point's side or on the device's.

- Device's side - Used for annoucing the device, its embedded devices and all services.
- Control Point's side - Used for keeping record of known devices.

There's no need to use the SSDP Server directly when using the control point or device API's.

4.3.1 SSDPServer class

Implementation of a SSDP Server.

Attributes

The attribute of the SSDPServer class are:

- server_name - name of the server.
- xml_description_filename - XML description filename.
- max_age - max age parameter, default is 1800.
- receive_notify - if False, ignores notify messages.
- known_devices - dict of the known devices.
- advertised - dict of the advertised devices.
- _callbacks - dict of the callbacks.

Methods

is_running()
Returns True if the SSDPServer is running, False otherwise.

start()
Starts the SSDPServer.

stop()
Sends bye bye notifications and stops the SSDPServer.

destroy()
Destroys the SSDPServer.

clear_device_list()
Clears the device list.

discovered_device_failed(*dev*)
Device could not be fully built, so forget it.

is_known_device(*usn*)
Returns if the device with the passed usn is already known.

subscribe(*name, callback*)
Subscribes a callback for an given name event.

unsubscribe(*name, callback*)
Unsubscribes a call back for an event.

announce_device()
Announces the device.

register_device(*device*)
Registers a device on the SSDP Server.

_datagram_received(*data, (host, pot)*)
Handles a received multicast datagram.

_discovery_request(*headers, (host, port)*)
Processes the discovery requests and responds accordingly.

_notify_received(*headers, (host, port)*)
Processes a presence announcement.

_register(*usn, st, location, server, cache_control, where='remote'*)
Registers a service or device.

_local_register(*usn, st, location, server, cache_control*)
Registers locally a new service or device.

_register_device(*device*)
Registers a device.

_renew_notifications()
Renew notifications (sends a notify).

_unregister(*usn*)
Unregisters a device.

_do_notify(*usn*)
Do a notification for the usn specified.

_do_byebye(*usn*)
Do byebye notification for the usn specified.

_callback(*name, *args*)
Performs callbacks for events.

_cleanup()
Cleans the SSDPServer by removing known devices and internal cache.

4.4 upnp_defaults — UPnP Constants

python-brisa provides a module which contains the UPnP constants defined on the 1.0 specification.

4.4.1 UPnPDefaults class

UPnP constants defined on the 1.0 specification

Constants

These are the constants holded by UPnPDefaults class

SERVICE_ID_PREFIX
Value: "urn:upnp-org:serviceId:"

SERVICE_ID_MS_PREFIX
Value: "urn:microsoft.com:serviceId:"

SCHEMA_VERSION
Value: "device-1-0"

SCHEMA_VERSION_MAJOR
Value: "1"

SCHEMA_VERSION_MINOR
Value: "0"

NAME_SPACE_XML_SCHEMA
Value: "urn:schemas-upnp-org:service-1-0"

SSDP_PORT
Value: 1900

SSDP_ADDR
Value: "239.255.255.250"

MSEARCH_DEFAULT_SEARCH_TIME
Value: 600.0

MSEARCH_DEFAULT_SEARCH_TYPE
Value: "ssdp:all"

4.5 UPnP Device

These are the UPnP modules of *python-brisa*. Each section describes a module and its usage.

4.5.1 `device` — Device classes

Device-side device class used for implementing and deploying UPnP devices.

Device class

Class that represents a device.

Attributes

Methods

4.5.2 `service` — Service classes

Implementing a simple service.

There is two ways of implement a service: using your own scpd.xml file or programatically describes it.

Implementing a service with a scpd.xml file.

Write your scpd.xml file.

```
<?xml version="1.0" encoding="utf-8"?>
<scpd xmlns="urn:schemas-upnp-org:service-1-0">
    <specVersion>
        <major>1</major>
        <minor>0</minor>
    </specVersion>
    <actionList>
        <action>
            <name>MyMethod</name>
            <argumentList>
                <argument>
                    <name>TextIn</name>
                    <direction>in</direction>
                    <relatedStateVariable>A_ARG_TYPE_Textin</relatedStateVariable>
                </argument>
```

```
                <argument>
                    <name>TextOut</name>
                    <direction>out</direction>
                    <relatedStateVariable>A_ARG_TYPE_Textout</relatedStateVariable>
                </argument>
            </argumentList>
        </action>
    </actionList>
    <serviceStateTable>
        <stateVariable sendEvents="no">
            <name>A_ARG_TYPE_Textout</name>
            <dataType>string</dataType>
        </stateVariable>
        <stateVariable sendEvents="yes">
            <name>A_ARG_TYPE_Textin</name>
            <dataType>string</dataType>
        </stateVariable>
    </serviceStateTable>
</scpd>
```

Now, create your service class and inherits from Service class. You will need to specify the service name, the service type, the scpd.xml file and implement a function to run a service action. The function name must begin with "soap" and ends with the action name:

```
from brisa.upnp.device.service import Service

service_name = 'MyService'
service_type = 'urn:schemas-upnp-org:service:MyService:1'

class MyService(Service):

    def __init__(self):
        Service.__init__(self, service_name, service_type, url_base='', scpd_xml_filepath='/path/to/

    def soap_MyMethod(self, *args, **kwargs):
    # Pay attention to the case sensitive arguments used here
    # and in the xml file you create for the service
    inArg = kwargs['TextIn']
    return {'TextOut': inArg + "Out!!"}
```

Implementing a service without a scpd.xml file.

You will need to specify your service definition programatically. Don't forget to specify at least one state variable and to set the "run_function" at each action. The "run_function" can have any name at this situation; it doesn't need to have the action name:

```
from brisa.upnp.device.service import Service
from brisa.upnp.device.action import Action, Argument
from brisa.upnp.device.service import StateVariable

service_name = 'MyService'
service_type = 'urn:schemas-upnp-org:service:MyService:1'

def MyMethod(*args, **kwargs):
    # Pay attention to the case sensitive arguments used here
```

```
    # and in the xml file you create for the service
    inArg = kwargs['TextIn']
    return {'TextOut': inArg + "Out!!"}

class MyService(Service):

    def __init__(self):
        Service.__init__(self, service_name, service_type, '')

        varIn = StateVariable(service=self, name="A_ARG_TYPE_Textin",
                              send_events=True, data_type="string")
        varOut = StateVariable(service=self, name="A_ARG_TYPE_Textout",
                               send_events=True, data_type="string")
        self.add_state_variable(varIn)
        self.add_state_variable(varOut)

        argIn = Argument(arg_name="TextIn", arg_direction=Argument.IN, arg_state_var=varIn)
        argOut = Argument(arg_name="TextOut", arg_direction=Argument.OUT, arg_state_var=varOut)

        actionMyMethod = Action(service=self, name="MyMethod", arguments=[argIn, argOut])
        actionMyMethod.run_function = MyMethod
        self.add_action(actionMyMethod)
```

4.6 UPnP Control Point

These are the UPnP modules of *python-brisa*. Each section describes a module and its usage.

4.6.1 `service` — Service classes

Subscribe for unicast eventing.

It's very simple to subscribe for service unicast eventing. You need to select the service object that you want to subscribe and call "event_subscribe" method.

There is a special parameter at "event_subscribe" method: auto_renew. If true, the framework will automatically renew the subscription before it expires. If false, the program need to call "event_renew" method before the subscription timeout. The renew callback will be used when auto_renew is true:

```
class ControlPoint(ControlPoint):

    def __init__(self):
        ControlPoint.__init__(self)

    def subscribe(self):
        service = self._select_service()
        service.event_subscribe(self.event_host, self._event_subscribe_callback, None, True, self._ev
        service.subscribe_for_variable("A_ARG_TYPE_Textin", self._event_callback)

    def _select_service(self):
        # Implement your way of find the service that you want
        return service

    def _event_subscribe_callback(self, cargo, subscription_id, timeout):
        print "Event subscribe done!"
```

```
        print 'Subscription ID: ' + str(subscription_id)
        print 'Timeout: ' + str(timeout)

    def _event_renew_callback(self, cargo, subscription_id, timeout):
        print "Event renew done!"
        print 'Subscription ID: ' + str(subscription_id)
        print 'Timeout: ' + str(timeout)

    def _event_callback(self, name, value):
        print "Event message!"
        print 'State variable:', name
        print 'Variable value:', value
```

Unsubscribe for unicast eventing.

In order to unsubscribe for service unicast eventing, you need to select the service object that you want to unsubscribe and call "event_unsubscribe" method:

```
def unsubscribe(self):
    service = self._select_service()
    service.event_unsubscribe(self.event_host, self._event_unsubscribe_callback, None)

def _event_unsubscribe_callback(self, cargo, old_subscription_id):
    print "Event unsubscribe done!"
    print 'Old subscription ID: ' + str(old_subscription_id)
```

CHAPTER

FIVE

APPLICATIONS

5.1 DBus Support

Both BRisa Media Server and BRisa Media Renderer applications support DBus, allowing them to integrate with a wide range of media systems. In this section we specify both interfaces.

5.1.1 BRisa Media Server

DBus specification

- Object path: `/br/edu/ufcg/embedded/brisa/MediaServer`
- Interface: `br.edu.ufcg.embedded.brisa.MediaServer`

Available Methods

Note: A watched folder is a folder whose files are being watched, that is, are served by the media server.

Note:

- DMS = DLNA Media Server
- CDS = Content Directory Subsystem or Service
- CM = Connection Manager service

`dms_halt()``:()`
Halts the media server.

`dms_get_server_info()``:()`
Returns a 7-tuple containing information about the device. The format is `(device version, brisa framework version, application version, server name, xbox compatibility, logging level, logging output)`.

`dms_reload_config()``:()`
Reloads the configuration from the database.

`dms_save_config()``:()`
Saves the runtime configuration.

`dms_set_xbox_compatibility(enable=True)``:()`
Enables or disables the XBox compatibility.

dms_cds_list_watched_audio_folders(): ()
Returns a list with the watched audio folders.

dms_cds_list_watched_video_folders(): ()
Returns a list with the watched video folders.

dms_cds_list_watched_picture_folders(): ()
Returns a list of the watched picture folders.

dms_cds_remove_watched_audio_folder(folder): ()
Removes an audio folder from being watched.

dms_cds_remove_watched_video_folder(folder): ()
Removes a video folder from being watched.

dms_cds_remove_watched_picture_folder(folder): ()
Removes a picture folder from being watched.

dms_cds_add_watch_audio_folder(folder): ()
Add an audio folder to be watched.

dms_cds_add_watch_video_folder(folder): ()
Add a video folder to be watched.

dms_cds_add_watch_picture_folder(folder): ()
Add a picture folder to be watched.

dms_cds_rescan_folders(): ()
Rescans all folders for changes (added or removed files).

dms_cds_browse(object_id, browse_flag, filter, starting_index, requested_count, sort_crite
Browses the CDS. Returns a DIDL XML string containing the result entries. The module **brisa.upnp.didl.didl_lite.Element** can be used for converting it into python objects.

dms_cds_search(container_id, search_criteria, filter, starting_index, requested_count, sort
Searches a CDS container for entries. Returns a DIDL XML string containing the result entries. The module **brisa.upnp.didl.didl_lite.Element** can be used for converting it into python objects.

dms_cds_get_search_caps(): ()
Retrieves the search capabilities of the CDS service.

dms_cds_get_sort_caps(): ()
Retrieves the sort capabilities of the CDS service.

dms_cds_get_system_update_id(): ()
Retrieves the CDS system update ID.

dms_cm_get_protocol_info(): ()
Retrieves the CM protocol info.

dms_cm_get_current_connection_ids(): ()
Retrieves CM current connection IDs.

dms_cm_get_current_connection_info(): ()
Retrieves CM current connection info.

5.1.2 BRisa Media Renderer

DBus specification

- Object path: `/br/edu/ufcg/embedded/brisa/MediaRenderer`

- Interface: `br.edu.ufcg.embedded.brisa.MediaRenderer`

Available Methods

Note:

- DMR = DLNA Media Renderer
- AVT = AVTransport service, where AV stands for Audio/Video
- RC = Render Control service
- CM = Connnection Manager service

dmr_halt():()
Halts the media renderer.

dmr_get_renderer_info():()
Returns a 6-tuple containing information about the device. The format is `(device version, brisa framework version, application version, renderer name, logging level, logging output)`.

dmr_avt_set_av_transport_uri(instance_id, current_uri, current_uri_metadata):()
Sets the AVT work URI (the one used for play(), pause(), stop(), and so on).

dmr_avt_get_media_info():()
Retrieves AVT media info.

dmr_avt_get_media_info_ext():()
Retrieves AVT extended media info.

dmr_avt_get_transport_info():()
Retrieves AVT transport info.

dmr_avt_get_position_info():()
Retrieves AVT position info.

dmr_avt_get_device_caps():()
Retrieves AVT device capabilities.

dmr_avt_get_transport_settings():()
Retrieves AVT transport settings.

dmr_avt_play():()
Starts the playback of the current URI.

dmr_avt_stop():()
Stops the playback.

dmr_avt_pause():()
Pauses the playback.

dmr_avt_seek(instance_id, unit, target):()
Seeks the playback.

dmr_avt_next():()
Plays next track (not implemented).

dmr_avt_previous():()
Plays previous track (not implemented).

dmr_rc_list_presets():()
Returns a list of RC presets.

dmr_rc_select_preset(preset):()
Select a preset for the RC.

dmr_rc_get_volume(instance_id, channel):()
Retrieves the volume of a channel.

dmr_rc_set_volume(instance_id, channel, desired_volume):()
Sets the volume of a channel.

dmr_cm_get_protocol_info():()
Retrieves CM protocol info.

dmr_cm_get_current_connection_info():()
Retrieves CM current connection info.

dmr_cm_get_current_connection_ids():()
Retrieves CM current connection IDs.

5.2 Writing plugins for BRisa Media Server

5.2.1 Introduction

BRisa Media Server can be easily extended to serve more content with plugins. We have a few plugins already implemented which can serve as code example:

- Flickr: enables access to Flickr pictures, such as user pictures or even featured ones.
- Youtube: enables access to Youtube user and featured videos.

Examples: **Media Library Plugin**.

5.2.2 Organization

Concerning files, your plugin must be placed in `$PREFIX/brisa_media_server/plugins/yourplugin`, where $PREFIX is usually `/usr/lib/python2.5/site-packages` or `/usr/local/lib/python2.6/dist-packages` and so on. On your `__init__.py` file you must import your plugin individual module, so that your plugin class is visible as subclass of our plugin interface (future plan is to have an automatic plugin install mechanism).

BRisa Media Server basic directory structure is composed only by a root folder, which belongs to what we call `RootPlugin`. When plugins are being loaded, they can add folders to the root folder or register themselves on other folders.

5.2.3 Interface

Plugins must be implemented using the `brisa.core.plugin.PluginInterface` interface.

Attributes that must be set during construction:

- `name`: plugin name
- **`usage`: True if your plugin is supposed to load, False otherwise. This is** usually loaded from a configuration file.

- **has_browse_filter: can be True or False. If your plugin** implements the browse() function with slicing/sorting capabilities (i.e. it uses `starting_index`, `requested_count`, `sort_criteria` parameters), this attribute must be `True`.

Methods that must be implemented (part of the interface):

PluginInterface.load():()
Loads the plugin.

PluginInterface.unload():()
Unloads the plugin.

PluginInterface.browse(object_id, browse_flag, filter, starting_index, requested_count, so
Browses plugin folders or items. Must return a list of DIDL objects (brisa.upnp.didl.didl_lite) representing the browse result.

PluginInterface.search(object_id, search_flag, filter, starting_index, requested_count, so
Searches the plugin folder for items.

If your plugin folder was correctly copied into `brisa_media_server/plugins/your_plugin` and on your file `/your_plugin/__init__.py` you import the module that contains your plugin class implementation, then the media server should load it automatically. For examples on that, please refer to our media_library example, currently on our *SVN repository <https://garage.maemo.org/svn/brisa>* under `trunk/app/media-server/src/plugins/media_library`.

Once your plugin is created, python-brisa automatically sets the `plugin_manager` attribute of your plugin, which points to our PluginManager. You will be using the PluginManager for retrieving the RootPlugin, which contains the root directory, for adding your plugin's own custom folders.

For retrieving the root plugin, consider the following code:

```
from brisa.core.plugin import PluginInterface


class MyPlugin(PluginInterface):

   (implement as the description we gave above)

   def load(self):
      # Retrieve the root plugin
      root_plugin = self.plugin_manager.root_plugin

      audio_container = root_plugin.get_container('Audio')

      if not audio_container:
            # Does not exist yet, add it
            audio_container = root_plugin.add_container('Audio')

      (...)
```

Specific implementation details of browse and search methods are better visualized on the media library example, as it uses everything that a plugin can do. In a nutshell, you should keep a dictionary of containers you added, keys being the container id attribute and values the container itself. When browsed, you query on this dictionary, and do the appropriate action that may vary from a plugin to another.

For instance, on a Flickr plugin we do not have actual files, but links to the web, so, you can cache the links on your containers (expensive) or just fetch the links when the browse() function tells it wants the contents of a container.

Plugins can also serve local files, which is also illustrated on the media library example.

CHAPTER

SIX

CODE EXAMPLES

These are some examples to show how simple it is to develop using BRisa

6.1 Creating a Binary Light Device

This is the implementation of a Binary Light Device, which the specifications you can find here.

The imports are few

```
from brisa.core.reactors import install_default_reactor
reactor = install_default_reactor()

import os

from brisa.upnp.device import Device, Service
```

This is everything you need to implement the device. First, you should now implement your Service class, doing the following:

```
class SwitchPower(Service):

    def __init__(self):
        Service.__init__(self, 'SwitchPower',
                         'urn:schemas-upnp-org:service:SwitchPower:1',
                         '',
                         os.getcwd() + '/SwitchPower-scpd.xml')
        self.target = False
        self.status = False

    def soap_SetTarget(self, *args, **kwargs):
        self.target = kwargs['newTargetValue']
        print 'Light switched ', {'1': 'on', '0': 'off'}.get(self.target, None)
        # Status here is the target because our device is "perfect" and
        # no errors could occur (this is a simulation)
        self.status = self.target
        return {}

    def soap_GetTarget(self, *args, **kwargs):
        return {'RetTargetValue': self.target}

    def soap_GetStatus(self, *args, **kwargs):
        return {'ResultStatus': self.status}
```

Ok, now your services are fine. Let's implement the device itself:

```
class BinaryLight(object):

    def __init__(self):
        self.server_name = 'Binary Light Device'
        self.device = None

    def _create_device(self):
        project_page = 'https://garage.maemo.org/projects/brisa'
        self.device = Device('urn:schemas-upnp-org:device:BinaryLight:1',
                             self.server_name,
                             manufacturer='Brisa Team. Embedded Laboratory '\
                                          'and INdT Brazil',
                             manufacturer_url=project_page,
                             model_name='Binary Light Device',
                             model_description='An UPnP Binary Light Device',
                             model_number='1.0',
                             model_url=project_page)

    def _add_services(self):
        switch = SwitchPower()
        self.device.add_service(switch)

    def start(self):
        self._create_device()
        self._add_services()
        self.device.start()
        reactor.add_after_stop_func(self.device.stop)
        reactor.main()


if __name__ == '__main__':
    device = BinaryLight()
    device.start()
```

Thats all it takes to implement the Binary Light Device.

You could follow these steps to implement any service/device, but remember that for now you'll have to write the service's xml yourself.

You can find this source code **Here** and the service's xml **Here**.

6.2 Creating a simple Control Point

This is an example of a simple Control Point, in this example we will implement a Binary Light command line Control Point, but this may help you to create any Control Point. This Control Point has the following features:

- Searching for devices
- Listing devices
- Handling events (new device located, removed device)

```
from brisa.core.reactors import install_default_reactor
reactor = install_default_reactor()
```

```
from brisa.core.network import parse_url
from brisa.core.threaded_call import run_async_function

from brisa.upnp.control_point.control_point import ControlPoint
```

Done the importing, lets implement the Control Point

```
service = ('u', 'urn:schemas-upnp-org:service:SwitchPower:1')
binary_light_type = 'urn:schemas-upnp-org:device:BinaryLight:1'

def on_new_device(dev):
    if not dev:
        return
    print 'Got new device:', dev.udn
    print "Type 'list' to see the whole list"


def on_removed_device(udn):
    print 'Device is gone:', udn
```

Event handling functions

```
def get_switch_service(device):
    return device.services[service[1]]


def create_control_point():
    c = ControlPoint()
    c.subscribe('new_device_event', on_new_device)
    c.subscribe('removed_device_event', on_removed_device)
    return c


def main():
    c = create_control_point()
    c.start()
    run_async_function(_handle_cmds, (c, ))
    reactor.add_after_stop_func(c.stop)
    reactor.main()
```

Now the command handler

```
def _handle_cmds(c):
    while True:
        try:
            input = raw_input('>>> ')
        except KeyboardInterrupt, EOFError:
            c.stop()
            break

        # Handle command
        if input == '':
            continue
        elif input == 'exit':
            break
        elif input == 'help':
```

```
        print 'Available commands: '
        for x in ['exit', 'help', 'search', 'set_light <dev number>',
                  'get_status', 'get_target', 'turn <on/off>', 'stop',
                  'list']:
            print '\t%s' % x
    elif input == 'stop':
        # Stop searching
        c.stop_search()
    elif input == 'search':
        # Start searching for devices of type upnp:rootdevice and repeat
        # search every 600 seconds (UPnP default)
        c.start_search(600, 'upnp:rootdevice')
    elif input == 'list':
        # List devices found
        k = 0
        for d in c.get_devices().values():
            print 'Device no.:', k
            print 'UDN:', d.udn
            print 'Name:', d.friendly_name
            print 'Device type:', d.device_type
            print 'Services:', d.services.keys() # Only print services name
            print 'Embedded devices:', [dev.friendly_name for dev in \
                d.devices.values()] # Only print embedded devices names
            print
            k += 1
    elif input.startswith('set_light'):
        try:
            c.current_server = devices[int(input.split(' ')[1])]
        except:
            print 'BinaryLight number not found. Please run list and '\
                  'check again'
            c.current_server = None
    elif input == 'get_status':
        try:
            service = get_switch_service(c.current_server)
            status_response = service.GetStatus()
            if status_response['ResultStatus'] == '1':
                print 'Binary light status is on'
            else:
                print 'Binary light status is off'
        except Exception, e:
            if not hasattr(c, 'current_server') or not c.current_server:
                print 'BinaryLight device not set.Please use set_light <n>'
            else:
                print 'Error in get_status():', e
    elif input.startswith('turn'):
        try:
            cmd = {'on': '1', 'off': '0'}.get(input.split(' ')[1], '')
            if not cmd:
                print 'Wrong usage. Please try turn on or turn off.'
                continue
            service = get_switch_service(c.current_server)
            service.SetTarget(newTargetValue=cmd)
            print 'Turning binary light', input.split(' ')[1]
        except Exception, e:
            if not hasattr(c, 'current_server') or not c.current_server:
                print 'BinaryLight device not set.Please use set_light <n>'
            else:
```

```
                print 'Error in set_status():', e
        elif input == 'get_target':
            try:
                service = get_switch_service(c.current_server)
                status_response = service.GetTarget()
                if status_response['RetTargetValue'] == '1':
                    print 'Binary light target is on'
                else:
                    print 'Binary light target is off'
            except Exception, e:
                if not hasattr(c, 'current_server') or not c.current_server:
                    print 'BinaryLight device not set.Please use set_light <n>'
                else:
                    print 'Error in get_target():', e

    # Stops the main loop
    reactor.main_quit()
```

And, finally to allow you to run it, add to the end

```
if __name__ == '__main__':
    main()
```

You can find this code **here**.

6.3 Creating a Qt Gui Device

This is very similar to create a simple device. You will need to create your Qt Gui, create the device, create the service and use the QtReactor:

```
from brisa.core.reactors.qtreactor import QtReactor
reactor = QtReactor()

from PyQt4 import QtCore, QtGui
from brisa.core import config
from brisa.upnp.device import Device
from brisa.upnp.device.service import Service, StateVariable

class QtDevice(QtGui.QWidget):

    def __init__(self):
        """ Constructor for class QtDevice, which constructs the Qt Gui.
        """
        QtGui.QWidget.__init__(self)

        self.verticalLayout = QtGui.QVBoxLayout(self)
        self.title = QtGui.QLabel("Qt Simple Device")
        font = QtGui.QFont()
        font.setPointSize(15)
        self.title.setFont(font)
        self.title.setAlignment(QtCore.Qt.AlignCenter)

        self.verticalLayout.addWidget(self.title)

        self.lineEdit = QtGui.QLineEdit(self)
```

```
        self.verticalLayout.addWidget(self.lineEdit)

        self.search_btn = QtGui.QPushButton("Start Device", self)
        self.verticalLayout.addWidget(self.search_btn)
        QtCore.QObject.connect(self.search_btn, QtCore.SIGNAL("clicked()"), self.start)

        self.stop_btn = QtGui.QPushButton("Stop Device", self)
        self.verticalLayout.addWidget(self.stop_btn)
        QtCore.QObject.connect(self.stop_btn, QtCore.SIGNAL("clicked()"), self.stop)

        self.lineEdit.setText('My Generic Device Name')
        self.root_device = None
        self.upnp_urn = 'urn:schemas-upnp-org:device:MyDevice:1'

    def _add_root_device(self):
        """ Creates the root device object which will represent the device
        description.
        """
        project_page = 'http://brisa.garage.maemo.org'
        serial_no = config.manager.brisa_version.replace('.', '').rjust(7, '0')
        self.root_device = Device(self.upnp_urn,
                                  str(self.lineEdit.text()),
                                  manufacturer='BRisa Team. Embedded '\
                                               'Laboratory and INdT Brazil',
                                  manufacturer_url=project_page,
                                  model_description='An Open Source UPnP generic '\
                                                    'Device',
                                  model_name='Generic Device Example',
                                  model_number=config.manager.brisa_version,
                                  model_url=project_page,
                                  serial_number=serial_no)

    def _add_services(self):
        """ Creates the root device service.
        """
        # Creating the example Service
        service_name = 'MyService'
        service_type = 'urn:schemas-upnp-org:service:MyService:1'

        myservice = Service(service_name, service_type, '')
        var = StateVariable(self, "A_ARG_TYPE_Variable",
                            True, False, "string")
        myservice.add_state_variable(var)

        # Inserting a service into the root device
        self.root_device.add_service(myservice)

    def _load(self):
        self._add_root_device()
        self._add_services()

    def start(self):
        """ Starts the root device.
        """
        self.stop()
        self._load()
        self.root_device.start()
        reactor.add_after_stop_func(self.root_device.stop)
```

```
    def stop(self):
        """ Stops the root device.
        """
        if self.root_device:
            self.root_device.stop()
            self.root_device = None

def main():
    qt_dev = QtDevice()
    qt_dev.show()
    reactor.main()

if __name__ == '__main__':
    main()
```

You can find this source code **Here**.

6.4 Creating a Qt Gui Control Point

This is very similar to create a simple control point. You will need to create your Qt Gui, create the control point and use the QtReactor:

```
from brisa.core.reactors.qtreactor import QtReactor
reactor = QtReactor()

from PyQt4 import QtCore, QtGui
from brisa.upnp.control_point import ControlPoint

class QtControlPoint(QtGui.QWidget):


    def __init__(self):
        """ Constructor for class QtControlPoint, which constructs the Qt Gui
        and the control point and starts it.
        """
        QtGui.QWidget.__init__(self)

        self.cp = ControlPoint()
        self.cp.subscribe('new_device_event', self.on_new_device)
        self.cp.subscribe('removed_device_event', self.on_removed_device)
        self.devices = []

        self.verticalLayout = QtGui.QVBoxLayout(self)
        self.title = QtGui.QLabel("Qt Simple Control Point")
        font = QtGui.QFont()
        font.setPointSize(15)
        self.title.setFont(font)
        self.title.setAlignment(QtCore.Qt.AlignCenter)

        self.verticalLayout.addWidget(self.title)

        self.search_btn = QtGui.QPushButton("Search", self)
        self.verticalLayout.addWidget(self.search_btn)
        QtCore.QObject.connect(self.search_btn, QtCore.SIGNAL("clicked()"), self.search)
```

```
        self.stop_btn = QtGui.QPushButton("Stop Search", self)
        self.verticalLayout.addWidget(self.stop_btn)
        QtCore.QObject.connect(self.stop_btn, QtCore.SIGNAL("clicked()"), self.stop_search)

        self.listView = QtGui.QListWidget(self)
        self.verticalLayout.addWidget(self.listView)

        self.cp.start()
        reactor.add_after_stop_func(self.cp.stop)

    def on_new_device(self, dev):
        if dev.udn not in [d.udn for d in self.devices]:
            self.devices.append(dev)
        self.list_devices()

    def on_removed_device(self, udn):
        for dev in self.devices:
            if dev.udn == udn:
                self.devices.remove(dev)
        self.list_devices()

    def list_devices(self):
        """ Lists the devices at list view.
        """
        self.listView.clear()
        for d in self.devices:
            self.listView.addItem(str(d.friendly_name))

    def search(self):
        """ Starts search for new devices.
        """
        self.cp.start_search(600, 'upnp:rootdevice')

    def stop_search(self):
        """ Stops search for new devices.
        """
        self.cp.stop_search()

def main():
    qt_cp = QtControlPoint()
    qt_cp.show()
    reactor.main()

if __name__ == '__main__':
    main()
```

You can find this source code **Here**.

CHAPTER

SEVEN

COPYRIGHT

CHAPTER

EIGHT

INDICES AND TABLES

- *Index*
- *Module Index*
- *Search Page*