Developer Documentation

Leandro Melo de Sales

André Dieb Martins

Embedded Systems and Pervasive Computing Laboratory

Developer Documentation

Leandro Melo de Sales
André Dieb Martins
Embedded Systems and Pervasive Computing Laboratory

Abstract

This guide describes the how to use the UPnP BRisa Framework. This document target audience are new UPnP developers, but it also may serve as reference for advanced ones.

Table of Contents

List of Figures

List of Examples

Chapter 1. Introduction

What is BRisa?

BRisa is an UPnP framework written in Python. Initially it focused on UPnP A/V specification, which concerns multimedia devices and control points. However, nowadays it has attained the status of a general UPnP Framework, providing facilities for creating UPnP devices and control points.

BRisa API comprehends internet messaging protocols and services (TCP, UDP, HTTP, SOAP), network utilitaries, threading management, logging, configurations, web server, UPnP control point and devices.

Even though it's not BRisa's main focus anymore, the BRisa project is still commited to maintaing and improving its UPnP A/V "branch", always complying to [ref_dlna] and UPnP specs.

Aside from the framework itself, BRisa also provides some applications, most concerning UPnP A/V:

- **brisa-media-server** - UPnP A/V Media Server 1.0 implementation with addition of a plugin architecture
- **brisa-media-server-plugins** - Plugins for BRisa Media Server (Youtube, Flickr, Shoutcast, Maemo Multimedia)
- **brisa-media-renderer** - UPnP A/V Media Renderer 1.0 implementation

Preliminary Reading

It is recommended that you have a basic understanding of Python programming language and of the UPnP specification. If you are already familiar with those, you can skip the first three items of Chapter 2, *Overview: UPnP & BRisa basics* and start reading from UPnP-UP.

Suggested documents:

- Python Documentation - http://docs.python.org/ [http://docs.python.org/]
- UPnP Device Architecture - http://upnp.org/specs/arch/UPnP-arch-DeviceArchitecture-v1.0-20080424.pdf [http://upnp.org/specs/arch/UPnP-arch-DeviceArchitecture-v1.0-20080424.pdf]
- UPnP MediaServer and MediaRenderer V1.0 - http://upnp.org/standardizeddcps/mediaserver.asp [http://upnp.org/standardizeddcps/mediaserver.asp]

Requirements

BRisa UPnP Framework (python-brisa) requires the following:

- Python 2.5.x, Python 2.5.x dev - http://python.org/
- Python-Cherrypy (>= 3.1.1) - http://www.cherrypy.org/ [http://www.cherrypy.org/]

brisa-media-server requires LightMediaScanner [http://lms.garage.maemo.org/], DBus/Python-Dbus [http://dbus.freedesktop.org/doc/dbus-python/doc/tutorial.html] and PyGTK [http://www.pygtk.org/].

brisa-media-renderer requires gstreamer0.10+ [http://gstreamer.freedesktop.org/downloap], python-gstreamer [http://gstreamer.freedesktop.org/modules/gst-python.html], gstreamer0.10-plugins-*

[http://gstreamer.freedesktop.org/download] and DBus/Python-Dbus [http://dbus.freedesktop.org/doc/dbus-python/doc/tutorial.html].

Integration

BRisa is supposed to run on the following systems:

- Linux
- Maemo

Concerning linux distributions, BRisa's been tested on Ubuntu Hardy and Gentoo 2008.0. Concerning the maemo platform, BRisa has been tested in N800 and N810. There are efforts to make BRisa Project work on Win32 platform.

Installing BRisa

Steps for installing BRisa (after all requirements are met):

Linux

- Get latest BRisa packages [https://garage.maemo.org/frs/?group_id=138] .
- For each package:

```
$ tar zxvf package.tar.gz
$ cd package
$ sudo python setup.py install
```

Maemo Devices

Single click install has not been updated yet for the latest release (0.7.0). It will be updated but at this moment BRisa for maemo is aimed mainly for developers with experience with ssh and scp. Thus, install instructions are the same as for above (Linux).

Chapter 2. Overview: UPnP & BRisa basics

BRisa framework can be divided into its python API and five end-user applications: ControlPoint, MediaServer and MediaRenderer, Launcher and ConfigurationTool. The python API is described thoroughly at Chapter 3, *BRisa packages* and its subsequent sections. It is also possible to consult the BRisa doc API [http://brisa.garage.maemo.org/apidoc/index.html].

In this section we will describe the roles of the UPnP end-user applications and their usage altogether inside UPnP.

UPnP

- MediaServer

  The media server role is to share UPnP multimedia items, such as audio files (.mp3, .wma, .ogg), video (.mpeg, .wmv, .swf) and image files (.gif, .jpg, .png), and remote multimedia streaming like Internet radio and online photo albums. These multimedia items can be stored in the local filesystem, in the local network hosts or even remotely spread on the Internet.

- MediaRenderer

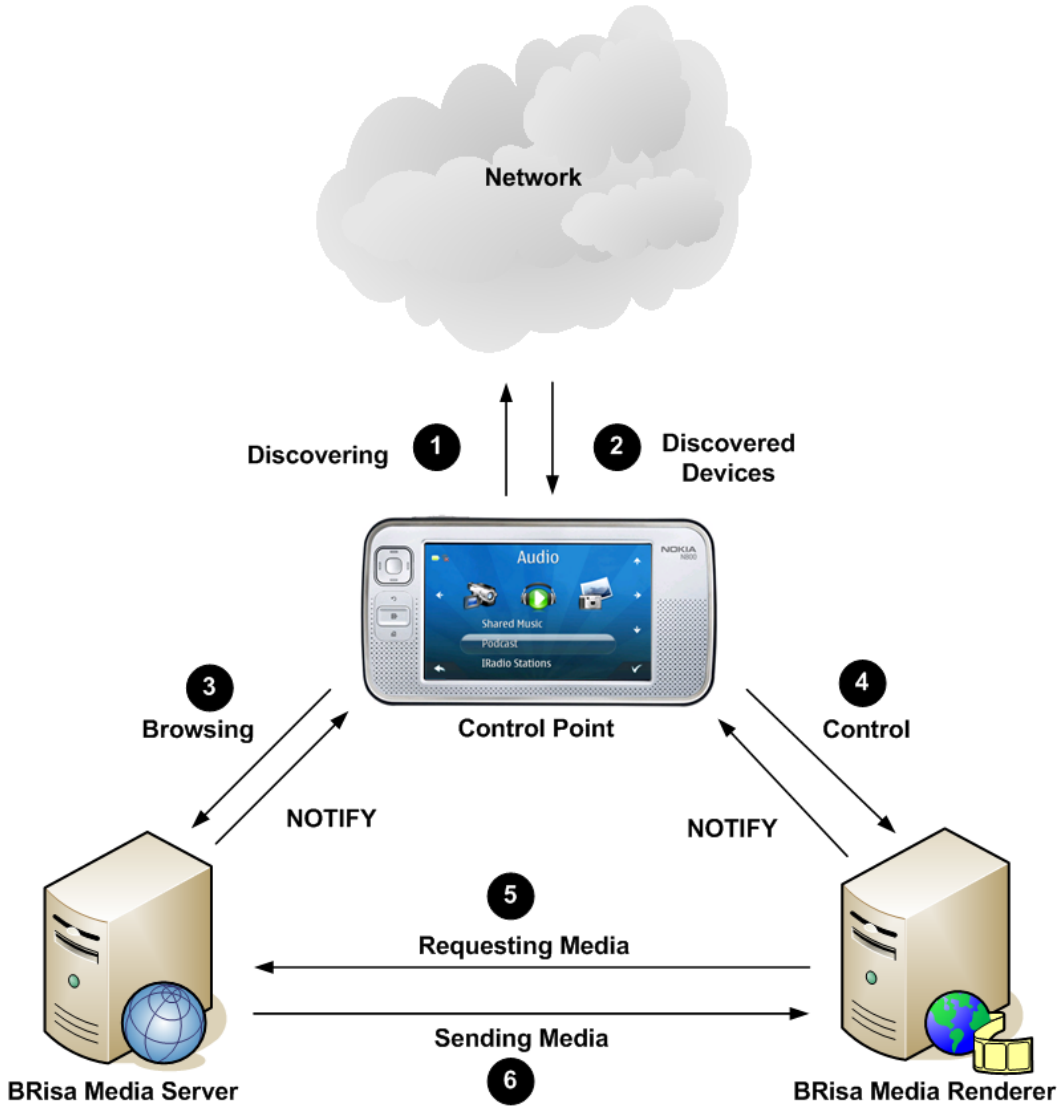
  The media render is capable of reproducing all multimedia data shared by any UPnP media server, including our media server implementation.

- ControlPoint

  Basically, control point is the tool that gives the user control over media servers and media renderers. When started, it sends a multicast message to the network asking for alive media servers and renderers. This procedure is called **MSearch** (Multicast Search) [ref_dlna].

  Each of the available servers in the network responses to the control point request, which at this point knows all of them. The user using the control point can browse any multimedia item at the media server and choose what media renderer he/she wants to render it. Figure 2.1, “Basic communication schema between UPnP devices. ” provides an illustration of this process.

Figure 2.1. Basic communication schema between UPnP devices.



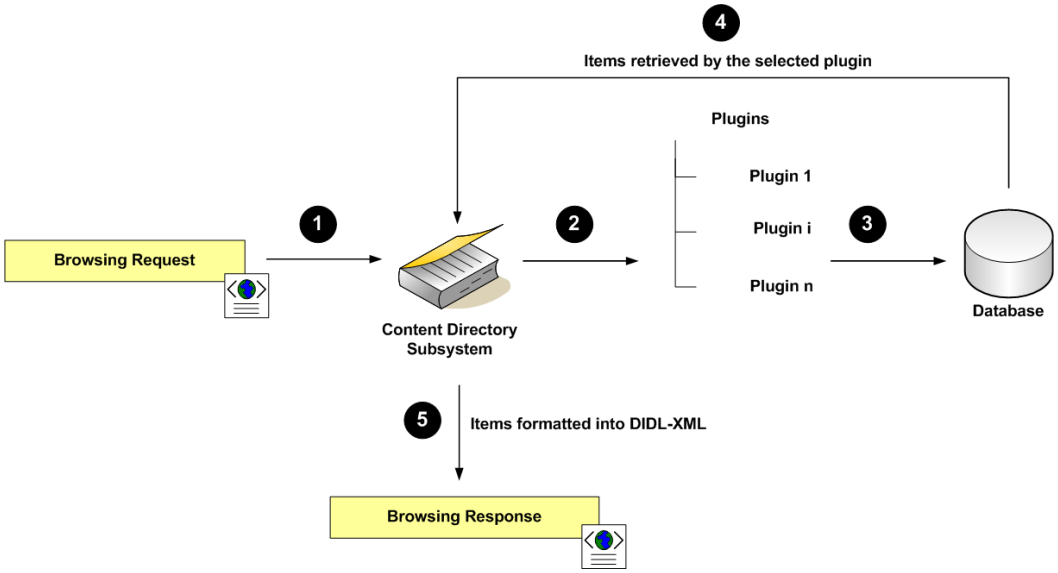
- BRisa MediaServer implementation

  BRisa's media server has been designed to have a dynamic asynchronous plugin-based architecture and a lightweight solution for querying UPnP items.

  We use a relational database (sqlite [http://www.sqlite.org/]) to store items and just when a browsing action comes from the network, the BRisa Content Directory Subsystem redirects the browsing to the specific plugin responsible for that browse action.

  The plugin then queries the requested items and returns them to the CDS. The CDS formats the items into a DIDL-XML specific format, which is used to represent complex digital objects. The formatted data is enveloped into a SOAP message and it is sent back to the corresponding control point. We present an illustration of this internal process in the Figure 2.2, “CDS and plugin-based architecture ”.

Figure 2.2. CDS and plugin-based architecture



Note that the plugin-based architecture is also shown in Figure 2.2, “CDS and plugin-based architecture ”. It enables third-party developers to implement CDS plugins that share multimedia data from a specific source. For more about plugin writing, please consult Chapter 9, *Services & Plugin Architecture*.

UPnP-UP

UPnP-UP is an extension of the UPnP specification that allows user authentication and authorization for UPnP devices and applications. The primary goal of this extension is to provide these features while being backwards compatible with pure UPnP. Currently BRisa implements a User Profile Server device and an API for UPnP-UP. For further information about UPnP-UP, go to the UPnP official website http://www.upnp-up.org

Figure 2.3. Communication schema between UPnP devices with UPnP-UP.

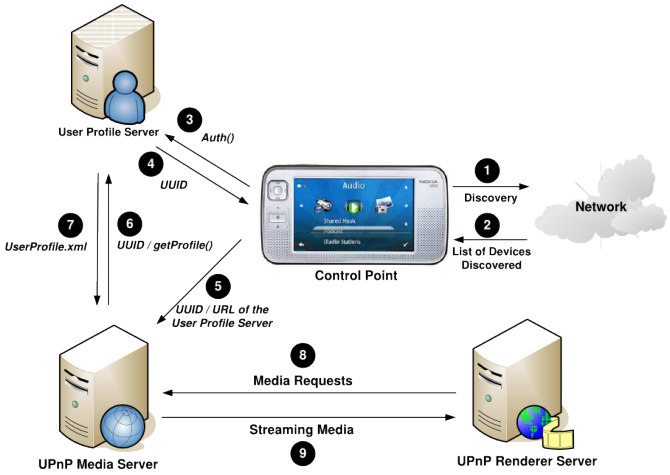
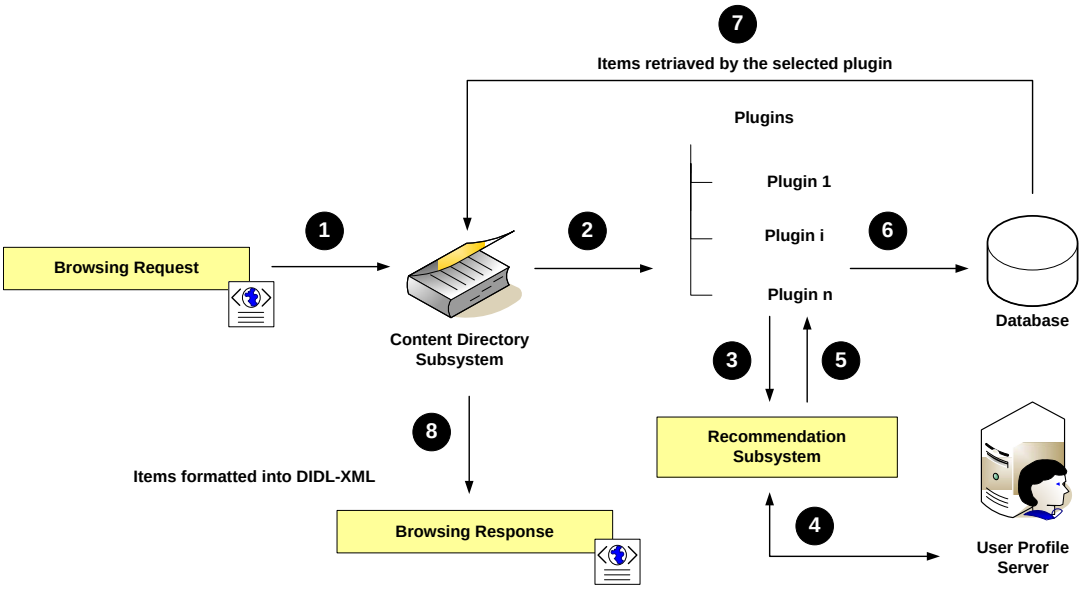


Figure 2.4. CDS and plugin-based architecture in UPnP-UP



Chapter 3. BRisa packages

Abstract

BRisa's package design principle is to try keeping related things together, but it is far from perfect. Please note that we are open to suggestions on the matter of package organization.

Packages can be summarized as follows:

- **brisa.config** - holds configuration tool and variables
- **brisa.control_point** - control point API
- **brisa.devices** - UPnP devices implemented
- **brisa.log** - logging API
- **brisa.threading** - thread management and thread related
- **brisa.services** - UPnP services implemented
- **brisa.upnp** - UPnP general
- **brisa.utils** - utility, networking, messaging, parsers
- **brisa.xml_descriptions** - holder package for UPnP xml descriptions

Description

- brisa.config

  This is where our BrisaConf class and some configuration variables are placed. This class is not supposed to be used directly, since its role on the configuration matter is only to check variables and set them on that package, so that you can use it later at any stage/module of your application.

- brisa.control_point

  - control_point.py - Base Control Point

    This module contains three functions and a ControlPoint class. More specifically:

    - **get_device_object** - makes and returns a description of a device object given the location
    - **get_device_object_async** - same as above, but asynchronous
    - **get_service_control_url** - returns a list of control_url given a service dict

    ControlPoint - core class for the API that implements an UPnP Control Point basic functionalities, such as device search control, subscribe/unsubscribe methods for device events and UPnP events.

  - control_point_av.py - UPnP A/V 1.0 ControlPoint implementation

    This module contains an UPnP A/V 1.0 ControlPoint implementation, which uses directly the Control Point API. It provides methods for controlling servers and renderers, such as **browse, search, get_search_capabilities, get_sort_capabilities, play, stop, pause, next, previous**. It's also an practical example for developers of how to use the control point API.

  - event and others

event.py contains the base for UPnP event listening. A GTK GUI implementation using our multimedia ControlPoint is also provided.

- brisa.devices

  Holder package for implemented devices. Three devices have been implemented:

  - **MediaServer** - device implementing a MediaServer, provides a simple start/stop interface *****
  - **MediaRenderer** - device implementing a MediaRenderer, provides a simple start/stop interface *****
  - **UserProfileServer** - device implementing a UserProfileServer ******

  ***** : Complies with UPnP MediaServer and MediaRenderer V1.0 [http://upnp.org/standardizeddcps/mediaserver.asp].

  ****** : User Profile - http://upnp-up.org/

- brisa.log

  Provides logging API with five different levels of logging. The level can be set at the **configuration file**.

  Levels list:

  - CRITICAL
  - ERROR
  - WARNING
  - DEBUG
  - INFO

- brisa.threading

  This package provides thread related classes and methods for managing them during execution (complete explanation of this can be found at Chapter 4, *Package threading: managed threading* of this document).

  It also provides a sleep function that assures blocking the whole sleeping time (python sleep is not completely safe [http://docs.python.org/lib/module-time.html]).

- brisa.services

  - avtransport

    Implementation of the AVTransport V 1.0 [http://upnp.org/standardizeddcps/documents/AVTransport1.0.pdf] service.

  - cds

    Implementation of the ContentDirectory V 1.0 [http://upnp.org/standardizeddcps/documents/ContentDirectory1.0.pdf] service. (cds) is the package which implements BRisa's plug-in-based architecture. For usage and plugin writing tips refer to the following sections.

  - connmgr

    Implementation of the ConnectionManager V 1.0 [http://upnp.org/standardizeddcps/documents/ConnectionManager1.0.pdf] service.

- gst_renderer

  Renderer service implementation using GStreamer [http://gstreamer.net].

- media_registrar_ms

- render_control

  Implementation of the RenderingControl V 1.0 [http://upnp.org/standardizeddcps/documents/RenderingControl1.0.pdf] service.

- web_server

  The web_server service provides a web server interface that allows publishing/removing resources and static files. Resources and static files are concepts of web servers but they can be abstracted:

  - **Static file**: object that matches with a file on the webserver. A request to an url that matches with a static file will have the file as response (for example, a simple download);

  - **Resource**: object that needs special information to be used for generating the response. For example, if you have a web form with login and password fields, the submit URL must be a Resource that understands the form data passed.

  For usage please refer to Chapter 9, *Services & Plugin Architecture*.

- brisa.upnp

  This package implements UPnP procedures (such as MSearch, SSDP server) and low level things (such as devices, services). It also contains UPnP constants, a DIDLLite implementation for UPnP containers & items and message handle related classes and objects (for handling device and service messages).

- brisa.utils

  This package provides useful functions and modules required by the UPnP implementation. It has high level implementations of network protocols (TCP, UDP) for listening and sending, network-related functions (getting your ip address, parsing urls) and a looping call. For usage please refer to the subsequent sections.

- brisa.xml_descriptions

  Contains scpd xml descriptions.

Chapter 4. Package threading: managed threading

This section explains how BRisa manages multiple threads and some concepts introduced by our multithread-based architecture.

Concepts

ThreadObject

ThreadObject is a feature of BRisa which abstracts Thread management from the developer. Basically, a ThreadObject is a Thread with some useful access/control methods and automatic registering/closure. When a ThreadObject is instantiated, it registers itself on the ThreadManager, who keeps track of all ThreadObjects for clean exiting.

Just as a normal Thread, all the logic of your Thread should be written in the **run()** method, which does nothing by default. If you need to do some actions before actually starting your ThreadObject routine, you should do this on the **prepare_to_start()** method. Analogous to the **prepare_to_start()** method, ThreadObject's also provides a **prepare_to_stop()** method, in which you should write actions to be done before exiting.

ThreadObject are named with the template **"ThreadObject %d: %s"** where %d is the unique ThreadObject number and %s is your class that inherits from ThreadObject. This keeps debugging easy.

Note that if you overwrite **stop()** or **start()** methods, both prepare_to_start and prepare_to_stop methods will become useless, since the logic for these calls will be lost (unless you copy it, which is not a very smart thing).

We **really recommend** users to write the **run()** method in a way that **is_running()** method is often read and handles the exiting of your ThreadObject. We still don't have a force-quit scheme implemented and by doing this the developer will avoid defunct processes. **DO NOT** modify the attribute *running* (it is part of the ThreadObject prepare_to_stop scheme and is automatically set when you call **stop()**).

Example 4.1. Using ThreadObject

```
from brisa.threading import ThreadObject

class MyThread(ThreadObject):
    def __init__(self, ...):
        ThreadObject.__init__(self, ...)

    def prepare_to_stop(self, ...):
        # Here my thread will do things necessary before stopping

    def prepare_to_start(self, ...):
        # Here my thread will do things necessary before starting

    def run(self, ...):
        # Here the thread will do its work!
        while self.is_running():
            # Work!
        # Got here after the a stop() call or run() finished
        # It is good here to close connections, for example.
```

ThreadManager

ThreadManager is the class that manages BRisa's threads and provides some useful functions such as running asynchronous function calls, a blocking main loop and proper threads stopping.

If you're using ThreadManager's loop and you want your threads to finish nicely, it is strongly recommended to make your threads inherit from brisa.threading.ThreadObject (see its documentation).

By doing this, your thread will automatically register itself on the ThreadManager and finish smoothly at the end of execution (ThreadManager().main_loop_quit()).

Using ThreadManager

- Using ThreadObject to have threads properly registered to the ThreadManager
- Running a main loop with **ThreadManager().main_loop()**
- Stopping all threads with **ThreadManager().stop_main_loop()**
- Registering "stop" functions to be called when the main loop quits
- Examples folder [https://garage.maemo.org/plugins/scmsvn/viewcvs.php/trunk/examples/?root=brisa]

ThreadManager is a **SINGLETON**. This means you can import it at any point of your program and use its functions.Some other useful management functions are also provided in the **brisa.threading** package.

ThreadedCall

ThreadedCall is a class that performs an asynchronous call and forwards the result of the call to a success callback in case of success or to a error callback in case some exception was raised. It can also perform delayed calls.

We strongly recommend a context analysis before using a ThreadedCall for avoiding race conditions.

Using ThreadedCall, run_async_function and run_async_call

Use **run_async_function()** for calls that you want to pass parameters in a tuple, like when using **thread.start_new_thread**.

Example 4.2. Using run_async_function()

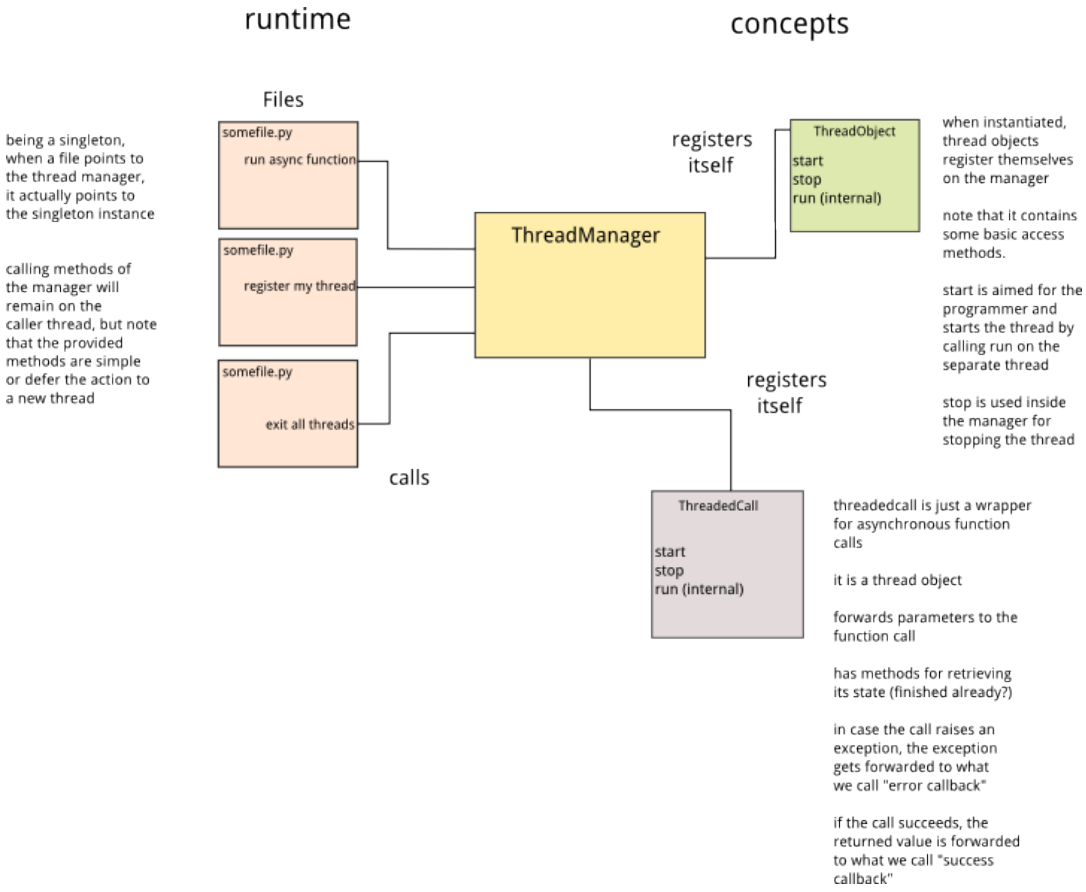
```
from brisa.threading import ThreadManager

def my_function(x, y, z, t):
    # Do something

ThreadManager.run_async_function(my_function, (5, 6, 7, 8))
```

For passing **args* and ***kwargs* to the function its easier to use ThreadedCall. It is possible to use it directly or using **run_async_call()** method. The difference is that **run_async_call()** makes the call and returns the control object (a ThreadedCall instance) and using the object you must call your ThreadedCall instance's **start()** method explicitly.

Example 4.3. brisa.threading.examples

Flow of Control

Figure 4.1. Flow of Control of ThreadManager, ThreadObject and ThreadedCall



Chapter 5. Configuration Usage

BRisa's configuration file is called **brisa.conf**. BRisa's has a template for it which is used as a base when installing BRisa. **Your own** configuration file is located at **~/.brisa/brisa.conf** and it holds your specific configuration choices (which you may have edited with the tool we provide or by hand).

Taking a quick look into **brisa.conf**, you may see its structure as configuration names between square brackets and key**:**value attributes below this name and before another one, as exemplified above:

```
[brisa]
servername = BRisa Media Server
renderername = BRisa Media Renderer
upservername = BRisa UP Server
version = 0.7
encoding = utf-8
xbox_compatible = on

[another name]
somename = somevalue
(...)
```

Almost all BRisa code uses the config package for retrieving variables. We may point out **brisaMediaServer.py**, **brisaMediaRenderer.py** and **brisaControlPoint.py** as main examples. As auxiliary examples we point **brisa.services.cds.plugins.filesystem.persistence.Persistence.py** where it uses the config module to retrieve a database URI (line 23).

Use Examples

Example 5.1. Encoding

```
from brisa import config

encoding = config.get_parameter("brisa", "encoding")
```

Example 5.2. Database URI

```
# brisa.conf configuration
[persistence]
connection = sqlite:/home/user/.brisa/filesystem.db

# python code for retrieving the above configuration
from brisa import config
database_uri = config.get_parameter('persistence',
 'connection').split('sqlite:')[1]
```

Chapter 6. Logger

For using the logger you just need to include the package and call the method corresponding to the level you want. For example, some log messages might make more sense in a DEBUG level instead of WARNING or INFO. The level choice is completely up to you.

Example 6.1. Using the logger

```
from brisa import log

log.debug("This is my debug message")
log.debug("This will appear as an INFO message!")
```

Chapter 7. Using ControlPoint API

At this point we suppose you already know the basics about the control point package. The most common use of the API is to create a control point class and inherit from our base ControlPoint. Here you'll find a simple example of a command line based control point that can search for devices and list them with some useful information. Although this example does not inherits from ControlPoint, it shows how to subscribe and get some useful info about devices (one can also try to reimplement this example with inheritance, as an exercise).

The second example has the same idea of the first but is GUI-based. Both examples are in the package python-brisa-examples [https://garage.maemo.org/plugins/scmsvn/viewcvs.php/trunk/python-brisa/examples/?root=brisa]

Creating a command line based ControlPoint

In this example we create a command line with the **raw_input()** function and we provide a few commands. Also, we have an Observer class that implements new/del device events. Note that we subscribe our Observer to listen for those events with the **subscribe()** method.

Example 7.1. Command line based ControlPoint

```
# Licensed under the MIT license
# http://opensource.org/licenses/mit-license.php or see LICENSE file.
# Copyright 2007-2008 Brisa Team <brisa-develop@garage.maemo.org>

import sys

from brisa.control_point.control_point import ControlPoint
from brisa.threading import ThreadManager


class Observer(object):
    # Needed callbacks
    l = []

    def on_new_device(self, dev):
        self.l.append(dev)

    def on_device_removed(self, udn):
        for dev in self.l:
            if dev.udn == udn:
                self.l.remove(dev)
                break

    def list_devices(self):
        for dev in self.l:
            print 'device:'
            print '\tudn:', dev.udn
            print '\tfriendly_name:', dev.friendly_name
            print '\tservices:', dev.services
            print '\ttype:', dev.device_type
            if dev.devices:
                print '\tchild devices:'
                for child_dev in dev.devices:
                    print '\t\tudn:', child_dev.udn
                    print '\t\tfriendly_name:', child_dev.friendly_name
                    print '\t\tservices:', dev.services
                    print '\t\ttype:', child_dev.device_type
            print


def main():
    print "BRisa's ControlPoint example\n"

    exit = False

    obs = Observer()
    cp = ControlPoint()
    cp.subscribe("new_device_event", obs.on_new_device)
    cp.subscribe("removed_device_event", obs.on_device_removed)

    try:
        while not exit:
            c = str(raw_input('>>> '))

            if c == 'exit':
                exit = True
            elif c == 'help':
                print 'commands: exit, help, start_search, stop_search, list'
            elif c == 'start_search':
                cp.start_search(600, "upnp:rootdevice")
                print 'search started!'
            elif c == 'stop_search':
                cp.stop_search()
                print 'search stopped!'
            elif c == 'list':
                print obs.list_devices()

            c = ''
    except KeyboardInterrupt, k:
        print 'quiting'

    ThreadManager().stop_all()
    sys.exit(0)


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

GUI based ControlPoint

We have also a GUI based ControlPoint example using GTK. It's simple and applies the same idea of the previous example. The code is a bit long and we prefer to give you the link [https://garage.maemo.org/plugins/scmsvn/viewcvs.php/trunk/python-brisa-examples/control_point_gui/?root=brisa]

Chapter 8. Diving into UPnP & BRisa

BRisa UPnP framework allows creating UPnP entities in a very simple manner. It permits the creation of UPnP services and devices, as well as facilitate the process of using the DIDL parsing. The following sections will guide you how to use the BRisa UPnP framework.

UPnP default constants

The module **brisa.upnp.upnp_defaults** contains UPnP default constants which you might want to use when using the framework. For example, it contains the default UPnP SSDP port and SSDP address for multicast messages.

Creating an UPnP Device

The basic steps for creating an UPnP device are:

- **Create a RootDevice** which holds information about your device and contains another devices
- **Set up an address for listening** using the WebServer
- **Create and add services** for your RootDevice and for its embedded devices
- **Set up a DeviceHandler** for announcing, stopping (sending bye-bye)
- **Publish your services and devices** on the WebServer

From this point on we will explain further these steps and show how to create a new device step by step. The code blocks below can be used to create a single program with little effort, but we split them for explaining purposes.

Warning: you can do these steps any way you want. The code below is just ONE way of creating a device.

Creating the RootDevice

The code below shows how to create and modify a RootDevice.

Example 8.1. Creating and modifying a RootDevice

```
from brisa.upnp.device import RootDevice

# I want my device to listen on this port
listen_url = 'http://localhost:7777'


def create_root_device(listen_url):
    # RootDevice(device_type, friendly_name, location)
    root_device = RootDevice('urn:schemas-upnp-
org:device:MyDevice:1',
                             'MyDevice',
                             listen_url)

    # Setting attributes
    root_device.manufacturer = 'Manufacturer'
    root_device.manufacturer_url = 'http://www.manufacturer.org'
    root_device.model_name = 'MyDevice version 1.0'
    root_device.model_number = '1.0'
    root_device.model_url = 'http://www.manufacturer.org/
mydevice/1.0/'
    root_device.serial_number = '0123456789ABCDEF'

    return root_device

(...)
```

The only difference between a Device and a RootDevice is that RootDevice can contain other devices. If you want to have embedded devices on the root, create them in the same way as the RootDevice and then add it to some RootDevice you have.

Creating services

In order to create a new service, you must have:

- **a Service object**, which contains information about the service, such as service location, UPnP URL's (control, event sub, presentation, scpd)
- **a ServiceControl object** which implements the methods of your service. These methods must be named in the form *soap_[method]()* (examples: soap_Browse, soap_GetSomething)

Code:

Example 8.2. Creating the Service

```
(...)

from brisa.upnp.service import Service

# Here we create the service that will have control at location
# listen_url/MyDevice/Control and the others just analogous.

my_service = Service(listen_url,
                     'MyDevice/control'
                     'MyDevice/event'
                     'MyDevice/presentation',
                     'MyDevice/scpd.xml')

(...)
```

Example 8.3. Creating the ServiceControl

```
(...)

from brisa.upnp.service import ServiceControl

class MyUPnPServiceControl(ServiceControl):

    # Required attribute that describes the namespace for the
 published methods
    namespace = ('u', 'urn:schemas-upnp-org:service:MyService:1')

    def __init__(self):
        ServiceControl.__init__(self)

    def start(self):
        pass

    def soap_MyMethod(self, *args, **kwargs):
        print 'Hello World!'

(...)
```

In this example we create a method called **MyMethod**, which will automatically be exported as a webservice when you add this service inside a Device/RootDevice object.

To add a service into a Device/RootDevice object, just append it using your object as shown below:

Example 8.4. Adding a service into a RootDevice

```
(...)

# Create the RootDevice
device = create_root_device()

control = MyUPnPServiceControl()

# Sets control of my_service to the control we want
my_service.control = control

# Adds the service to the device
device.services[my_service.service_type] = my_service

(...)
```

The next section provides further details regarding to the creation of a new BRisa UPnP device.

DeviceHandler

DeviceHandler configures some internal things for your device.

Example 8.5. Creating the device handler

```
(...)

from brisa.upnp.upnp_handler import DeviceHandler

# Parameter is a name for your server
device_handler = DeviceHandler('MyDevice Server')

# Configure my device (which is a RootDevice)
(xml_name, xml_path) = device_handler.config_device(device)

# The device handler created a xml with information about my device

(...)
```

Finishing up settings

The only thing left to do is to set up our webserver and create some management methods for starting and stopping.

Example 8.6. Set up the webserver

```
(...)

from brisa.threading import ThreadManager
from brisa.services.web_server import WebServer, StaticFile

webserver = WebServer()

# Sets the webserver's listen url to the one I want for my device
 and services
webserver.listen_url = listen_url

# Add the information about my device on the webserver (xml file)
webserver.add_static_file(xml_name, StaticFile(xml_path))

# Start service, returns a Resource correspondent with the Service
service_resource = my_service.start_service()

# Adds the resource of the service to the webserver
webserver.add_resource(my_service.friendly_name, service_resource)


def stop():
    self.device_handler.stop_device()
    return True

def start():
    # Start webserver
    webserver.start()

    # Start the service control
    my_service.control.start()

    # Enables my device receiving requests
    device_handler.start_device()

    # Register stop function
    ThreadManager().register_stop_function(stop)

    # Blocking main loop. Captures exit signals (SIGTERM, SIGINT)
    ThreadManager().main_loop()


if __name__=="__main__":
    start()
```

Chapter 9. Services & Plugin Architecture

Using the web server

BRisa's web server service is delivered as a **Singleton**. For using it, just retrieve the singleton and add anything you want using the provided methods. For understanding the examples below it's recommended that you're already familiar with the package.

Example 9.1. Serving a file

```
from brisa.services.web_server import WebServer, StaticFile

webserver = WebServer()
webserver.listen_url = 'http://url:port'

my_file = StaticFile('/path/to/my_file')

# I want my file to be found at http://url:port/my_file_name
webserver.add_static_file('my_file_name', my_file)

webserver.start()
```

Example 9.2. Using a Resource

```
from brisa.services.web_server import WebServer, Resource

webserver = WebServer()
webserver.listen_url = 'http://url:port'


class MyResource(Resource):

    def render(self, uri, request, response):
        # The answer to the request is the return of this
        # function.
        return 'My resource has been requested!'

    def getChildWithDefault(self, uri, params):
        # In this function you will return who you want to
        # render the request. You can have some logic with
        # uri and params that matches with who you want to
        # render the request. Here we're returning the
        # class itself because this is what we want.
        return self

# I want my resource to be located at http://url:port/my_resource'
web_server.add_resource('my_resource', MyResource())

web_server.start()
```

Brief explanation of the content directory asynchronous plugin structure

Here we assume that you already read the short description about cds.

When writing plugins, developers must place their package on the *services/cds/plugins* folder. They will be automatically loaded according to their plugin configuration. From this point on you will learn about the following:

Writing your own plugin

Using the Plugin class

The Plugin class is the base class for all plugin. It is located at module brisa.services.cds.plugin. When inheriting from it, it is not needed to pass any additional parameters, but you must implement the following methods:

- **load** - load implementation for your plugin
- **unload** - unload implementation for your plugin
- **browser** - browse implementation for your plugin
- **search** - search implementation for your plugin

If you do not implement any of these four methods and a request arrives for your plugin and for that specific method, an exception will be raised remembering you to implement it.

You may name your plugin by renaming the *name* attribute and if your **browser** method implements internally the advanced search features (sorting, slicing), you must set the *has_browse_filter* to True.

BRisa has some plugins implemented, such as Filesystem, Youtube, Shoutcast, and etc. You may refer to them as practical examples [https://garage.maemo.org/plugins/scmsvn/viewcvs.php/trunk/python-brisa/brisa/services/cds/plugins/?root=brisa].

Notes on plugin configurations

If you want your plugin to provide some configuration, you can add an entry at brisa.conf for it. Note that when you're writing your plugin, you will need to use brisa's configuration module to retrieve or set the values you have on brisa.conf.

Our implemented plugins may serve as examples of this. The Chapter 5, *Configuration Usage* contains a plugin configuration example. For our Filesystem plugin we have an entry on brisa.conf we called persistence. It contains a few "key = value" fields which we load when loading this plugin.

Chapter 10. Utility

How to use TCP and UDP listeners and senders

BRisa provides TCP and UDP listeners, called respectivelly TCPListener and UDPListener, both located at the module brisa.utils.network_listeners. These listeners share the same interface, and for using it just create an object of it and subscribe for listening. You can subscribe with a single callback (data_callback, passed for the constructor) or by creating a listener and subscribing with **subscribe(listener)** method. Your listener must implement the **data_received(data, addr)** method. **addr** is a tuple with the form (host, port).

```
TCPListener(port, interface='', listeners=[], data_callback=None,
            shared_socket=None)
    Creates a TCP connection listener on the port and interface
 specified.
    Forwards data listened to data_callback and to the listeners
 (that must
    implement a data_received() method). Can also reuse a socket
 connection
    already open, which in this case must be passed on the
 shared_socket
    parameter.

UDPListener(addr, port, interface='', listeners=[],
 data_callback=None,
            shared_socket=None):
    Creates an UDP connection listener on the port, address and
 interface
    specified. Forwards data listened to data_callback and to the
 listeners
    (that must implement the data_received() method). Can also
 reuse a socket
    connection already open, which in this case must be passed on
 the
    shared_socket parameter.
```

Concerning TCP and UDP senders, respectively TCPTransport and UDPTransport, the only thing needed is to create an object and use the provided methods (all non-blocking).

```
UDPTransport(ttl=2)
    Creates an UDP packet sender with the specified TTL.

    set_TTL(ttl)
        Sets the TTL

    send_data(data, (host, port))
        Sends data to the host and port specified.

    send_delayed(delay, data, (host, port))
        Sets a timer for sending the data after the specified
 delay.
```

```
TCPTransport()
    Creates an TCP packet sender.

    send_data(data, (host, port))
        Connects and sends data to the host and port specified.

    connect_and_feed(feeder, (host, port))
        Connects and feeds the connection with the feeder until it
 finishes the
        feeding. Feeder is supposed to be a generator.
```

get_ip_address - Getting your IP address

Retrieves the ip address on the given interface.

```
get_ip_address(interface_name)
    Returns a string with the ip address on the specified
 interface.
```

get_active_ifaces - Getting available network interfaces

Returns a list of the active network interfaces.

```
get_active_ifaces()
    Return a list of the active network interfaces
    Default route of /proc/net/route has the destination field set
 to 00000000
```

parse_url - Parsing an url

parse_url is just an alias for the urlparse.urlparse function.

```
parse_url(url)
    Parse a URL into 6 components:
    scheme://netloc/path;params?query#fragment
    Return a 6-tuple: (scheme, netloc, path, params, query,
 fragment).
    Note that we don't break the components up in smaller bits
    (e.g. netloc is a single string) and we don't expand %
    escapes.
```

http_call - Sending HTTP requests

Returns a HTTPResponse object for the given call.

```
http_call(method, url, body='', headers={})
```

```
    Performs a HTTP request and returns an HTTPResponse associated
 with the given
    call.

    method: HTTP method (NOTIFY, POST, etc...)
    url: target URL
    body: body of the message
    headers: additional headers
```

LoopingCall - Performing repeated function calls in specified intervals

As mentioned on the title, LoopingCall performs repeated function calls in a specified interval.

```
LoopingCall(function, *args, **kwargs)

    start(interval, now=True)
        Starts the LoopingCall with the specified interval. If now
 is not True,
        it waits the interval once before starting the loop.

    stop()
        Stops the LoopingCall.
```

Note that **args** and **kwargs** are the parameters you want to pass to your function in each call.

Generating properties

```
gen_property_with_default(name, fget=None, fset=None, doc="")
    Generates a property of a name either with a default fget or a
 default fset.

gen_property_of_type(name, _type, doc="")
    Generates a type-forced property associated with a name.
 Provides type
    checking on the setter (coherence between value to be set and
 type
    specified).
```

Chapter 11. Contact

After reading this document, if you have any doubts feel free to contact us on irc or through the mailing lists.

- IRC
  - **#brisa** *@ irc.freenode.org*
- Mailing lists
  - brisa-develop - Subscribe [http://garage.maemo.org/mailman/listinfo/brisa-develop]
  - brisa-discuss - Subscribe [http://garage.maemo.org/mailman/listinfo/brisa-discuss]

References

[ref_dlna] *DLNA Specification. http://www.dlna.org* .