Developer's Guide: Adding a new ConPaaS service

January 17, 2012

1 Introduction

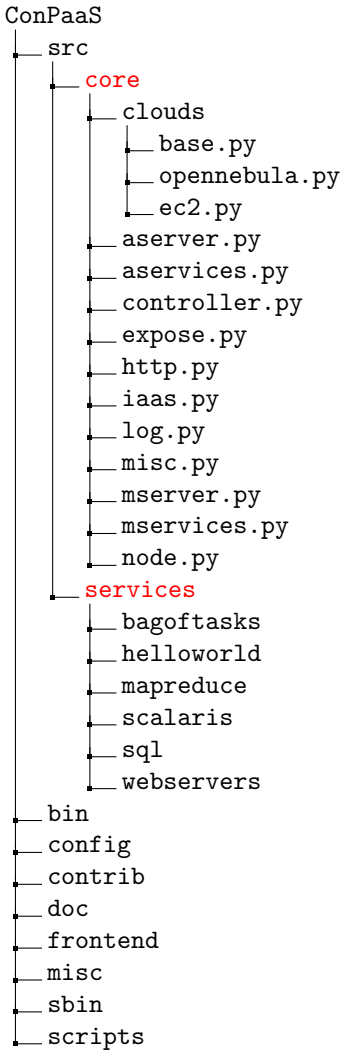
A ConPaaS service may consist of three main entities: the manager, the agent and the frontend. The (primary) manager resides in the first VM that is started by the frontend when the service is created and its role is to manage the service by providing supporting agents, maintaining a stable configuration at any time and by permanently monitoring the service's performance. An agent resides on each of the other VMs that are started by the manager. The agent is the one that does all the work. Note that a service may contain one manager and multiple agents, or multiple managers that also act as agents.

To implement a new ConPaaS service, you must provide a new manager service, a new agent service and a new frontend service (we assume that each ConPaaS service can be mapped on the three entities architecture). To ease the process of adding a new ConPaaS service, we propose a framework which implements common functionality of the ConPaaS services. So far, the framework provides abstraction for the IaaS layer (adding support for a new cloud provider should not require modifications in any ConPaaS service implementation) and it also provides abstraction for the HTTP communication (we assume that HTTP is the preferred protocol for the communication between the three entities).

1.1 ConPaaS directory structure

You can see bellow the directory structure of the ConPaaS software. The *core* folder under *src* contains the ConPaaS framework. Any service should make use of this code. It contains the manager hhtp server, which instantiates the python manager class that implements the required service; the agent http server that instantiates the python agent class (if the service requires agents); the iaas abstractions and other useful code.

A new service should be added in a new python module under the *ConPaaS/src/services* folder.



In the next paragraphs we describe how to add the new ConPaaS service.

2 Service's name

The first step in adding a new ConPaaS service is to choose a name for it. This name will be used to construct, in a standardized manner, the file names of the scripts required by this service (see below). Therefore, the names should not contain spaces, nor unaccepted characters.

3 Scripts

To function properly, ConPaaS uses a series of configuration files and scripts. Some of them must be modified by the administrator, i.e. the ones concerning the cloud infrastructure, and the others are used, ideally unchanged, by the manager and/or the agent. A newly added service would ideally function with

the default scripts. If, however, the default scripts are not satisfactory (for example the new service would need to start something on the VM, like a memcache server) then the developers must supply a new script/config file, that would be used instead of the default one. This new script's name must be preceded by the service's chosen name (as described above) and will be selected by the frontend at run time to generate the contextualization file for the manager VM. (If the frontend doesn't find such a script/config file for a given service, then it will use the default script). **Note that some scripts provided for a service do not replace the default ones, intead they will be concatenated to them (see below the agent and manager configuration scripts).**
Below we give an explanation of the scripts and configuration files used by a ConPaaS service (there are other configuration files used by the frontend but are not relevant to the ConPaaS service). Basically there are two scripts that a service uses to boot itself up - the manager contextualization script, which is executed after the manager VM booted, and the agent contextualization script, which is executed after the agent VM booted. These scripts are composed of several parts, some of which are customizable to the needs of the new service.
In the ConPaaS home folder ($CONPAAS_HOME) there is the *config* folder that contains configuration files in the INI format and the *scripts* folder that contains executable bash scripts. Some of these files are specific to the cloud, other to the manager and the rest to the agent. These files will be concatenated in a single contextualization script, as described below.

- Files specific to the Cloud:

  (1) $CONPAAS_HOME/config/cloud/*cloud_name*.cfg, where *cloud_name* refers to the clouds supported by the system (for now opennebula and ec2). So there is one such file for each cloud the system supports. These files are filled in by the administrator. They contain information such as the username and password to access the cloud, the OS image to be used with the VMs, etc. These files are used by the frontend and the manager, as both need to ask the cloud to start VMs.

  (2) $CONPAAS_HOME/scripts/cloud/*cloud_name*, where *cloud_name* refers to the clouds supported by the system (for now opennebula and ec2). So, as above, there is one such file for each cloud the system supports. These scrips will be included in the contextualization files. For example, for opennebula, this file sets up the network.

- Files specific to the Manager:

  (3) $CONPAAS_HOME/scripts/manager/manager-setup, which prepares the environment by copying the ConPaaS source code on the VM, unpacking it, and setting up the PYTHONPATH environment variable.

  (4) $CONPAAS_HOME/config/manager/*service_name*-manager.cfg, which contains configuration variables specific to the service manager (in INI format). If the new service needs any other variables (like a path to a file in the source code), it should provide an annex to the default manager config file. This annex must be named *service_name*-manager.cfg and will be concatenated to default-manager.cfg

(5) $CONPAAS_HOME/scripts/manager/*service_name*-manager-start, which starts the server manager and any other programs the service manager might use.

(6) $CONPAAS_HOME/sbin/manager/*service_name*-cpsmanager (will be started by the *service_name*-manager-start script), which starts the manager server, which in turn will start the requested manager service.

Scripts (1), (2), (3), (4) and (5) will be used by the frontend to generate the contextualization script for the manager VM. After this scripts executes, a configuration file containing the concatenation of (1) and (4) will be put in $ROOT_DIR/config.cfg and then (6) is started with the config.cfg file as a parameter that will be forwarded to the new service.

Examples:

Listing 1: Script (1) - ConPaaS/config/cloud/opennebula.cfg

```
[iaas]
DRIVER = OPENNEBULA

URL = http://opennebula.domain:4567
USER = user
PASSWORD = password
IMAGE_ID = 3
INST_TYPE = 4
NET_ID = 1
NET_GATEWAY = 192.168.122.1
NET_NAMESERVER = 192.168.122.1
```

Listing 2: Script (2) - ConPaaS/scripts/cloud/opennebula

```
#!/bin/bash
# Copyright (C) 2010-2011 Contrail consortium.
#
# This file is part of ConPaaS, an integrated runtime environment
# for elastic cloud applications.
#
# ConPaaS is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as ↩
    published by
# the Free Software Foundation, either version 3 of the License, ↩
    or
# (at your option) any later version.
#
# ConPaaS is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public ↩
    License
# along with ConPaaS.  If not, see <http://www.gnu.org/licenses↩
    />.

# This is part of the contextualization script
# for the VMs started on Opennebula.
# The rest is generated by the frontend.

if [ -f /mnt/context.sh ]; then
  . /mnt/context.sh
fi

/sbin/ifconfig eth0 $IP_PUBLIC
/sbin/ip route add default via $IP_GATEWAY
echo "nameserver $NAMESERVER" > /etc/resolv.conf
```

Listing 3: Script (3) - ConPaaS/scripts/manager/manager-setup

```
#!/bin/bash

# Ths script is part of the contextualization file. It
# copies the source code on the VM, unpacks it, and sets
# the PYTHONPATH environment variable.

# Is filled in by the frontend
FRONTEND=%FRONTEND_URL%
SOURCE=$FRONTEND/code
ROOT_DIR=/root
CPS_HOME=$ROOT_DIR/ConPaaS

wget -P $ROOT_DIR/ $SOURCE/ConPaaS.tar.gz
tar -zxf $ROOT_DIR/ConPaaS.tar.gz -C $ROOT_DIR/
export PYTHONPATH=$CPS_HOME/src/:$CPS_HOME/contrib/
```

Listing 4: Script (4) - ConPaaS/config/manager/default-manager.cfg

```
[manager]

# Service TYPE will be filled in by the frontend
TYPE = %CONPAAS_SERVICE_TYPE%

BOOTSTRAP = $SOURCE

# These three are used by the manager to instruct the frontend to
# decrement the number of credits the user has.
# They are used when a VM ran more than 1 hour.
# Everything will be filled in by the frontend
FE_SERVICE_ID = %CONPAAS_SERVICE_ID%
FE_CREDIT_URL = %FRONTEND_URL%/callback/decrementUserCredit.php
FE_TERMINATE_URL = %FRONTEND_URL%/callback/terminateService.php

# This directory structure already exists in the VM (with ROOT = ↩
    '') - see
# the 'create new VM script' so do not change ROOT unless you ↩
    also modify
# it in the VM. Use these files/directories to put variable data ↩
    that
# your manager might generate during its life cycle
ROOT =
LOG_FILE = %(ROOT)s/var/log/cpsmanager.log
ETC = %(ROOT)s/etc/cpsmanager/
VAR_TMP = %(ROOT)s/var/tmp/cpsmanager/
VAR_CACHE = %(ROOT)s/var/cache/cpsmanager/
VAR_RUN = %(ROOT)s/var/run/cpsmanager/
CODE_REPO = %(VAR_CACHE)s/code_repo

CONPAAS_HOME = $CPS_HOME

# Add below other config params your manager might need and save ↩
    a file as
# %service_name%-manager.cfg
# Otherwise this file will be used by default
```

Listing 5: Script (5) - ConPaaS/scripts/manager/default-manager-start

```
#!/bin/bash

# This script is part of the contextualization file. It
# starts a python script that parses the given arguments
# and starts the manager server, which in turn will start
# the manager service.

# This file is the default manager-start file. It can be
# customized as needed by the sevice.
```

```
$CPS_HOME/sbin/manager/default-cpsmanager -p 80 -c $ROOT_DIR/↩
    config.cfg 1>$ROOT_DIR/manager.out 2>$ROOT_DIR/manager.err &
manager_pid=$!
echo $manager_pid > $ROOT_DIR/manager.pid
```

Listing 6: Script (6) - ConPaaS/sbin/manager/default-cpsmanager

```
#!/usr/bin/python
'''
Copyright (C) 2010-2011 Contrail consortium.

This file is part of ConPaaS, an integrated runtime environment
for elastic cloud applications.

ConPaaS is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published↩
     by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

ConPaaS is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with ConPaaS.  If not, see <http://www.gnu.org/licenses/>.


Created on Jul 4, 2011

@author: ielhelw
'''
from os.path import exists
from conpaas.core.mserver import ManagerServer

if __name__ == '__main__':
  from optparse import OptionParser
  from ConfigParser import ConfigParser
  import sys

  parser = OptionParser()
  parser.add_option('-p', '--port', type='int', default=80, dest=↩
      'port')
  parser.add_option('-b', '--bind', type='string', default='↩
      0.0.0.0', dest='address')
  parser.add_option('-c', '--config', type='string', default=None↩
      , dest='config')
  options, args = parser.parse_args()

  if not options.config or not exists(options.config):
    print >>sys.stderr, 'Failed to find configuration file'
    sys.exit(1)

  config_parser = ConfigParser()
  try:
    config_parser.read(options.config)
  except:
    print >>sys.stderr, 'Failed to read configuration file'
    sys.exit(1)

  """
  Verify some sections and variables that must exist in the ↩
      configuration file
  """
  config_vars = {
    'manager': ['TYPE', 'BOOTSTRAP', 'LOG_FILE',
                'FE_CREDIT_URL', 'FE_TERMINATE_URL', '↩
                    FE_SERVICE_ID'],
    'iaas': ['DRIVER'],
  }
  config_ok = True
```

```
  for section in config_vars:
    if not config_parser.has_section(section):
      print >>sys.stderr, 'Missing configuration section "%s"' % ↩
          (section)
      print >>sys.stderr, 'Section "%s" should contain variables ↩
          %s' % (section, str(config_vars[section]))
      config_ok = False
      continue
    for field in config_vars[section]:
      if not config_parser.has_option(section, field)\
      or config_parser.get(section, field) == '':
        print >>sys.stderr, 'Missing configuration variable "%s" ↩
            in section "%s"' % (field, section)
        config_ok = False
  if not config_ok:
    sys.exit(1)

  """
  Start the manager server
  """
  print options.address, options.port
  d = ManagerServer((options.address, options.port),
                    config_parser,
                    reset_config=True)
  d.serve_forever()
```

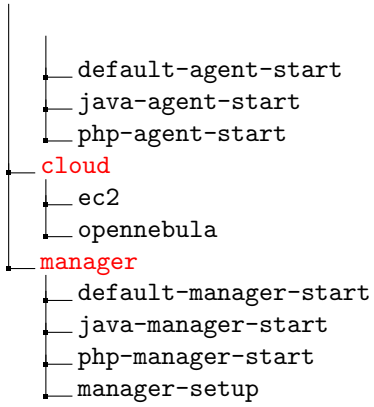
- Files specific to the Agent

  They are similar to the files described above for the manager, but this time the contextualization file is generated by the manager.

3.1 Scripts and config files directory structure

Bellow you can find the directory structure of the scripts and configuration files described above.

```
ConPaaS
├── config
│   ├── agent
│   │   ├── default-manager.cfg
│   │   ├── java-manager.cfg
│   │   └── php-manager.cfg
│   ├── cloud
│   │   ├── ec2.cfg
│   │   ├── opennebula.cfg
│   │   └── clouds-template.cfg
│   └── manager
│       └── default-manager.cfg
├── sbin
│   ├── agent
│   │   ├── default-cpsmagent
│   │   ├── java-cpsagent
│   │   └── php-cpsagent
│   └── manager
│       ├── default-cpsmanager
│       └── php-cpsmanager
└── scripts
    └── agent
        └── agent-setup
```



4 Implementing a new ConPaaS service

In this section we describe how to implement a new ConPaaS service by providing an example which can be used as a starting point. The new service is called *helloworld* and will just generate helloworld strings. Thus, the manager will provide a method, called get_helloworld which will ask all the agents to return a 'helloworld' string (or another string chosen by the manager).
We will start by implementing the agent. We will create a class, called HelloWorldAgent, which implements the required method - get_helloworld, and put it in *conpaasservices/helloworld/agent/agent.py* (Note: make the directory structure as needed and providing empty __init__.py to make the directory be recognized as a module path). As you can see in Listing 7, this class uses some functionality provided in the conpaas.core package. The conpaas.core.expose module provides a python decorator (@expose) that can be used to expose the http methods that the agent server dispatches. By using this decorator, a dictionary containing methods for http requests GET, POST or UPLOAD is filled in behind the scenes. This dictionary is used by the built-in server in the conpaas.core package to dispatch the HTTP requests. The module conpaas.core.http contains some useful methods, like HttpJsonResponse and HttpErrorResponse that are used to respond to the HTTP request dispatched to the corresponding method. In this class we also implemented a method called startup, which only changes the state of the agent. This method could be used, for example, to make some initializations in the agent. We will describe later the use of the other method, check_agent_process.

Listing 7: conpaas/services/helloworld/agent/agent.py

```
from conpaas.core.expose import expose
from conpaas.core.http import HttpJsonResponse, HttpErrorResponse,\
                        HttpFileDownloadResponse, HttpRequest,\
                        FileUploadField, HttpError, _http_post
from conpaas.core.log import create_logger

class HelloWorldAgent():
    def __init__(self,
                 config_parser, # config file
                 **kwargs):     # anything you can't send in ↩
                     config_parser
                                # (hopefully the new service won't ↩
                                    need anything extra)

      self.logger = create_logger(__name__)
```

```
        self.state = 'INIT'
        self.gen_string = config_parser.get('agent', 'STRING_TO_GENERATE↩
            ')

    @expose('GET')
    def check_agent_process(self, kwargs):
        """Check if agent process started - just return an empty ↩
            response"""
        if len(kwargs) != 0:
            return HttpErrorResponse('ERROR: Arguments unexpected')
        return HttpJsonResponse()

    @expose('POST')
    def startup(self, kwargs):
        self.state = 'RUNNING'
        self.logger.info('Agent started up')
        return HttpJsonResponse()

    @expose('GET')
    def get_helloworld(self, kwargs):
        if self.state != 'RUNNING':
            return HttpErrorResponse('ERROR: Wrong state to get_helloworld↩
                ')
        return HttpJsonResponse({'result':self.gen_string})
```

Let's assume that the manager wants each agent to generate a different string. The agent should be informed about the string that it has to generate. To do this, we could either implement a method inside the agent, that will receive the required string, or specify this string in the configuration file with which the agent is started. We opted for the second method just to illustrate how a service could make use of the config files and also, maybe some service agents/managers need some information before having been started.
Therefore, we will provide the *helloworld-agent.cfg* file (see Listing 8) that will be concatenated to the default-manager.cfg file. It contains a variable ($STRING) which will be replaced by the manager.

Listing 8: ConPaaS/config/agent/helloworld-agent.cfg

```
STRING_TO_GENERATE = $STRING
```

Now let's implement an http client for this new agent server. See Listing 9. This client will be used by the manager as a wrapper to easily send requests to the agent. We used some useful methods from conpaas.core.http, to send json objects to the agent server.

Listing 9: conpaas/services/helloworld/agent/client.py

```
from conpaas.core.http import _jsonrpc_get, _jsonrpc_post, _http_post
import httplib, json

def _check(response):
  code, body = response
  if code != httplib.OK: raise Exception('Received http response code ↩
      %d' % (code))
  data = json.loads(body)
  if data['error']: raise Exception(data['error'])
  else: return data['result']

def check_agent_process(host, port):
  method = 'check_agent_process'
  return _check(_jsonrpc_get(host, port, '/', method))

def startup(host, port):
  method = 'startup'
```

```
  return _check(_jsonrpc_post(host, port, '/', method))

def get_helloworld(host, port):
  method = 'get_helloworld'
  return _check(_jsonrpc_get(host, port, '/', method))
```

Next, we will implement the manager in the same manner: we will write the *HelloWorldManager* class and place it in the file *conpaas/services/helloworld/-manager/manager.py*. To make use of the IaaS abstractions, we need to instantiate a Controller which controls all the requests to the clouds on which ConPaaS is running. Note the lines:

```
1: self.controller = Controller( config_parser)
2: self.controller.generate_context('helloworld')
```

The first line instantiates a Controller. The controller maintains a list of cloud objects generated from the *config_parser* file. There are several functions provided by the controller which are documented in the doxygen documentation of file *controller.py*. The most important ones, which are also used in the Hello World service implementation, are: *generate_context* (which generates a template of the contextualization file); *update_context* (which takes the contextualization template and replaces the variables with the supplied values); *create_nodes* (which asks for additional nodes from the specified cloud or the default one) and *delete_nodes* (which deletes the specified nodes).
Note that the *create_nodes* function accepts as a parameter a function (in our case *check_agent_process*) that tests if the agent process started correctly in the agent VM. If an exception is generated during the calls to this function for a given period of time, then the manager assumes that the agent process didn't start correctly and tries to start the agent process on a different agent VM.

Listing 10: conpaas/services/helloworld/manager/manager.py

```
from threading import Thread, Lock, Timer, Event

from conpaas.core.expose import expose
from conpaas.core.controller import Controller
from conpaas.core.http import HttpJsonResponse, HttpErrorResponse,\
                        HttpFileDownloadResponse, HttpRequest,\
                        FileUploadField, HttpError, _http_post
from conpaas.core.log import create_logger
from conpaas.services.helloworld.agent import client

class HelloWorldManager(object):

    # Manager states - Used by the frontend
    S_INIT = 'INIT'         # manager initialized but not yet started
    S_PROLOGUE = 'PROLOGUE' # manager is initializing
    S_RUNNING = 'RUNNING'   # manager is running
    S_ADAPTING = 'ADAPTING' # manager is in a transient state - ↩
        frontend will keep
                            # polling until manager out of transient ↩
                               state
    S_EPILOGUE = 'EPILOGUE' # manager is shutting down
    S_STOPPED = 'STOPPED'   # manager stopped
    S_ERROR = 'ERROR'       # manager is in error state

    def __init__(self,
                 config_parser, # config file
                 **kwargs):     # anything you can't send in ↩
                     config_parser
                                # (hopefully the new service won't ↩
                                   need anything extra)
```

```
        self.config_parser = config_parser
        self.logger = create_logger(__name__)
        self.logfile = config_parser.get('manager', 'LOG_FILE')
        self.state = self.S_PROLOGUE
        self.nodes = []
        # Setup the clouds' controller
        self.controller = Controller(config_parser)
        self.controller.generate_context('helloworld')

    @expose('POST')
    def startup(self, kwargs):
        self.logger.info('Manager started up')
        self.state = self.S_RUNNING
        return HttpJsonResponse()

    @expose('POST')
    def shutdown(self, kwargs):
        self.state = self.S_EPILOGUE
        Thread(target=self._do_shutdown, args=[]).start()
        return HttpJsonResponse()

    def _do_shutdown(self):
        self.controller.delete_nodes(self.nodes)
        self.state = self.S_STOPPED
        return HttpJsonResponse()

    @expose('POST')
    def add_nodes(self, kwargs):
        self.controller.update_context(dict(STRING='helloworld'))
        if self.state != self.S_RUNNING:
            return HttpErrorResponse('ERROR: Wrong state to add_nodes'↩
                )
        if not 'count' in kwargs:
            return HttpErrorResponse('ERROR: Required argument doesn\'↩
                t exist')
        if not isinstance(kwargs['count'], int):
            return HttpErrorResponse('ERROR: Expected an integer value↩
                 for "count"')
        count = int(kwargs.pop('count'))
        self.state = self.S_ADAPTING
        Thread(target=self._do_add_nodes, args=[count]).start()
        return HttpJsonResponse()

    def _do_add_nodes(self, count):
        node_instances = self.controller.create_nodes(count, \
                                            client.check_agent_process,↩
                                                 5555)
        self.nodes += node_instances
        # Startup agents
        for node in node_instances:
            client.startup(node.ip, 5555)
        self.state = self.S_RUNNING
        return HttpJsonResponse()

    @expose('GET')
    def list_nodes(self, kwargs):
        if len(kwargs) != 0:
            return HttpErrorResponse('ERROR: Arguments unexpected')
        if self.state != self.S_RUNNING:
            return HttpErrorResponse('ERROR: Wrong state to list_nodes↩
                ')
        return HttpJsonResponse({
              'helloworld': [ node.vmid for node in self.nodes ],
              })

    @expose('GET')
    def get_service_info(self, kwargs):
        if len(kwargs) != 0:
            return HttpErrorResponse('ERROR: Arguments unexpected')
        return HttpJsonResponse({'state': self.state, 'type': '↩
            helloworld'})

    @expose('POST')
    def remove_nodes(self, kwargs):
```

```
        if self.state != self.S_RUNNING:
            return HttpErrorResponse('ERROR: Wrong state to ↩
                remove_nodes')
        if not 'count' in kwargs:
            return HttpErrorResponse('ERROR: Required argument doesn\'↩
                t exist')
        if not isinstance(kwargs['count'], int):
            return HttpErrorResponse('ERROR: Expected an integer value↩
                 for "count"')
        count = int(kwargs.pop('count'))
        self.state = self.S_ADAPTING
        Thread(target=self._do_remove_nodes, args=[count]).start()
        return HttpJsonResponse()

    def _do_remove_nodes(self, count):
        for i in range(0, count):
            self.controller.delete_nodes([self.nodes.pop(0)])
        return HttpJsonResponse()

    @expose('GET')
    def get_helloworld(self, kwargs):
        if self.state != self.S_RUNNING:
            return HttpErrorResponse('ERROR: Wrong state to ↩
                get_helloworld')
        # Just get_helloworld from all the agents
        for node in self.nodes:
            data = client.get_helloworld(node.ip, 5555)
            self.logger.info('Received %s from %s', data, node.vmid)
        return HttpJsonResponse({
            'helloworld': [ node.vmid for node in self.nodes ],
            })

    @expose('GET')
    def getLog(self, kwargs):
        if len(kwargs) != 0:
            return HttpErrorResponse('ERROR: Arguments unexpected')
        try:
            fd = open(self.logfile)
            ret = ''
            s = fd.read()
            while s != '':
              ret += s
              s = fd.read()
              if s != '':
                  ret += s
            return HttpJsonResponse({'log': ret})
        except:
            return HttpErrorResponse('Failed to read log')
```

We can also implement a client for the manager server (see Listing 11). This will allow us to use the command line interface to send requests to the manager, if the frontend integration is not available.

Listing 11: conpaas/services/helloworld/manager/client.py

```
import httplib, json
from conpaas.core.http import HttpError, _jsonrpc_get, _jsonrpc_post, ↩
    _http_post, _http_get

def _check(response):
  code, body = response
  if code != httplib.OK: raise HttpError('Received http response code ↩
      %d' % (code))
  data = json.loads(body)
  if data['error']: raise Exception(data['error'])
  else: return data['result']

def get_service_info(host, port):
  method = 'get_service_info'
  return _check(_jsonrpc_get(host, port, '/', method))
```

```
def get_helloworld(host, port):
  method = 'get_helloworld'
  return _check(_jsonrpc_get(host, port, '/', method))

def startup(host, port):
  method = 'startup'
  return _check(_jsonrpc_post(host, port, '/', method))

def add_nodes(host, port, count=0):
  method = 'add_nodes'
  params = {}
  params['count'] = count
  return _check(_jsonrpc_post(host, port, '/', method, params=params))

def remove_nodes(host, port, count=0):
  method = 'remove_nodes'
  params = {}
  params['count'] = count
  return _check(_jsonrpc_post(host, port, '/', method, params=params))

def list_nodes(host, port):
  method = 'list_nodes'
  return _check(_jsonrpc_get(host, port, '/', method))
```

The last step is to register the new service to the conpaas core. One entry must be added to files *conpaas/core/mservices.py* and *conpaas/core/aservices.py*, as it is indicated in Listing 12 and Listing 13. Because the java and php services use the same code for the agent, there is only one entry in the agent services (*aservices.py*), called *web* which is used by both the webservices.

Listing 12: conpaas/core/mservices.py

```
'''
This file contains all the available manager services implementations.
'''

services = {'php'    : {'class' : 'PHPManager',
                        'module': 'conpaas.services.webservers.manager↩
                            .internal.php'},
            'java'   : {'class' : 'JavaManager',
                        'module': 'conpaas.services.webservers.manager↩
                            .internal.java'},
            'helloworld' : {'class' : 'HelloWorldManager',
                        'module': 'conpaas.services.helloworld.manager↩
                            .manager'}

           }
```

Listing 13: conpaas/core/aservices.py

```
'''
This file contains all the available services implementations.
'''

services = {'web' : {'class' : 'WebServersAgent',
                     'module': 'conpaas.services.webservers.↩
                         agent.internals'},
            'helloworld' : {'class' : 'HelloWorldAgent',
                     'module': 'conpaas.services.helloworld.↩
                         agent.agent'}
           }
```

5 Integrating the new service with the frontend

So far there is no easy way to add a new frontend service. Each service may require distinct graphical elements. In this section we explain how the Hello World frontend service has been created.

5.1 Manager states

As you have noticed in the Hello World manager implementation, we used some standard states, e.g. INIT, ADAPTING, etc. By calling the *get_service_info* function, the frontend knows in which state the manager is. Why do we need these standardized stated? As an example, if the manager is in the ADAPTING state, the frontend would know to draw a loading icon on the interface and keep polling the manager.

5.2 Files to be modified

```
frontend
└── www
    ├── create.php
    └── lib
        └── service
            └── factory
                └── __init__.php
```

Several lines of code must be added to the two files above for the new service to be recognized. If you look inside these files, you'll see that knowing where to add the lines and what lines to add is self-explanatory.

5.3 Files to be added

```
frontend
├── www
│   └── helloworld.php
├── lib
│   ├── service
│   │   └── helloworld
│   │       └── __init__.php
│   └── ui
│       └── instance
│           └── helloworld
│               └── __init__.php
└── images
    └── helloworld.png
```