OpenRenderManagement Documentation

Release 1.7.3

Arnaud Chassagne, Jerome Samson

December 02, 2014

Contents

CHAPTER 1

TODO - User guide

1.1 Introduction

1.2 General Architecture

1.3 Monitoring jobs and render nodes

1.4 Submitting on the farm

CHAPTER 2

TODO - Admin Guide

2.1 Prequisites

2.2 Installing a server

2.3 Installing a worker

CHAPTER 3

IN PROGRESS - API reference

3.1 Using PULI API

Several guides have been written on the project wiki to present how to use the puliclient API.

3.2 Graph

class `puliclient.`**`Graph`**

Data structure to submit to Puli server.
It describes one or several tasks that will be executer on the renderfarm.

`__init__`(*name*, *root=None*, *user=None*, *poolName='default'*, *maxRN=-1*, *tags={}*)

Create a new graph object with given name and parameters.
If root is given, it will be attached to the graph (wether it is a task or a taskgroup).
It root is not specified a default taskgroup will be created as the root node.

Parameters

- **name** – a string describing the graph
- **root** – optionnal node to be attached to the graph
- **user** – the owner of the graph
- **poolName** – a pool of rendernodes to use for this execution
- **maxRN** – a max number of concurrent rendernodes to use

Raises GraphError

`add`(*pElem*)

Adds a task or a taskgroup to the graph's root.
If the graph root is a task, an error is raised.

Parameters **pElem** (*Task or TaskGroup*) – the node to attach to the graph

Returns a reference to the attached node

Raise GraphError

addChain (*pEdgeChain, pEndStatusList=[3]*)

Create edges to the current graph.
Edges are given as a list of element to link as a chain.
Example chain: [taskA, taskB, taskC, ...]
Elements to chain can be either Tasks or TaskGroups.
A connection error is raised if the edge chain or status list are
not properly formatted.

Parameters

- **pEdgeList** – List of elements indicating nodes to be chained in the given order
- **pEndStatusList** – A list of end status to be defined for every connections

Returns a boolean indicating if the connections have been executed corretly

Raise ConnectionError

addEdges (*pEdgeList*)

Create edges to the current graph.
Edges are given as a list of element, with each elem being a
sequence of: sourceNode, destNode and status
example edge list: [(taskA, taskB), (taskA, taskC, [ERROR]), ...]

Parameters **pEdgeList** – List of elements indicating the source and dest node and a list of ending status (source, desti, [endStatus])

Returns a boolean indicating if the connections have been executed corretly

Raise ConnectionError

addList (*pElemList*)
Add a list of nodes to the graph's root.

Parameters **pElemList** – list of nodes (task or taskgroup)

Raise GraphError

addNewCallable (*targetCall, name='', user_args=(), user_kwargs={}, **kwargs*)

Wraps around TaskGroup method to add a new Task. It accepts any callable
to run on the renderfarm. Additionnal Task arguments can be added as keyword args

Parameters

- **targetCall** – callable to be serialized and run on the render farm
- **name** – optionnal task name (if not defined, method or function name will be used)
- **user_args** – a list or tuple representing formal arguments to use with the callable
- **user_kwargs** – a dict representing keyword arguments to use with the callable

addNewTask (**args*, ***kwargs*)

Creates a new task and attach it to the graph's root.
If the graph root is a task, an error is raised.

Parameters

- **args** – standard Task arguments
- **kwargs** – keyword Task arguments

Returns a reference to the attached node

Raise GraphError

addNewTaskGroup (**args*, ***kwargs*)

Creates a new taskgroup and attach it to the graph's root.
If the graph root is a task, an error is raised

Parameters

- **args** – standard TaskGroup arguments
- **kwargs** – keyword TaskGroup arguments

Returns a reference to the attached node

Raise GraphError

execNode (*pCommand*)

Emulate the execution of a command on a worker node.
2 possible execution mode: with a subprocess or direct
- Calls the "commandwatcher.py" script used by the worker process to keep a similar behaviour
  Command output and error messages are left in stdout/stderr to give the user a proper feedback of its command
- Create CommandWatcherObject in current exec

Parameters **pCommand** – dict containing the command description and arguments

Raise GraphExecInterrupt when a keyboard interrupt is raised

execute ()

Prepare a graph representation to execute locally.
The following steps will be executed :

```
1. Prepare the graph representation with GraphDumper
2. Parse the representation to extract all commands in a single list in "id" order
   Several attribute of the task are stored with each command (taksid, end, dependencies...)
3. While there are some "ready" command
  3.1 Parse ready commands
        Execute command
  3.2 Parse blocked commands
        Check dependencies and increment "nbReadyAfterCheck" counter
```

```
    3.3 If nbReadyAfterCheck == 0
          BREAK the while loop
  4. Write summary and return
```

Returns the final state of the graph

Return type int

Raise GraphExecError

prepareGraphRepresentation()

Prepare a graph representation to be sent to the server or executed
locally. Several steps must be taken:
- parse graph to resolve dependencies on taskgroups
- parse graph to expand/decompose tasks and taskgroups

submit(*host='puliserver'*, *port=8004*)

Prepare a graph representation and send it to the server.
- prepare graph
- use GraphDumper class to serialize it into a JSON representation
- submit data via http

Parameters

- **str** (*host*) – server name to connect to
- **int** (*port*) – server port to connect to

Returns A tuple with the server response ie. ('SERVER_URL/nodes/Id', 'Graph created. Created nodes: ...')

Raise GraphSubmissionError

3.3 Taskgroup

class puliclient.**TaskGroup**

A node of a graph that can contain other nodes: taskgroups or tasks

__init__(*name*, *expander=None*, *arguments={}*, *tags={}*, *environment={}*, *timer=None*, *priority=0*, *dispatchKey=0*)

addNewCallable(*targetCall*, *name=''*, *user_args=()*, *user_kwargs={}*, ***kwargs*)

Wraps around TaskGroup method to add a new Task. It accepts any callable
to run on the renderfarm. Additionnal Task arguments can be added as keyword args
It uses the internal "CallableRunner" to reload arguments and execute the callable.

Parameters

- **targetCall** – callable to be serialized and run on the render farm
- **name** – optionnal task name (if not defined, method or function name will be used)

- **user_args** – a list or tuple representing positionnal arguments to use with the callable
- **user_kwargs** – a dict representing keyword arguments to use with the callable

addNewTask (**args, **kwargs*)

Creates a task with given args and add it to the current TaskGroup.

Parameters

- **args** – standard Task arguments
- **kwargs** – keyword Task arguments

Returns a reference on the created item

Raise GraphError if task creation failed

addNewTaskGroup (**args, **kwargs*)

Creates a task group with given args and add it to the TaskGroup.

Parameters

- **args** – standard Task arguments
- **kwargs** – keyword Task arguments

Returns a reference on the created item

Raise GraphError if task group creation failed

addTask (*task*)

Add the task given as parameter to the current TaskGroup.

Parameters **task** – task object to add to the hierarchy

addTaskGroup (*taskGroup*)

Add the taskgroup given as parameter to the current TaskGroup.

Parameters **taskGroup** – a taskgroup to add to the hierarchy

classmethod createFromTask (*task*)

Creates a taskgroup from a task given in parameter

Parameters **task** – task model

Returns a new taskgroup

dependsOn (*task, statusList=[3]*)

Create a dependency constraint between the current node and the given
task or taskGroup at a particular status.

Parameters

- **task** – a Task or a TaskGroup to dependsOn
- **statusList** – a list of statuses to be reached (any of it) to validate the dependency

expand (*hierarchy*)

Expands a taskgroup hierarchy.
- first expand itself
- then expand or decompose its children (taskgroup or tasks)

Parameters **hierarchy** – the hierarchy root node

Returns a reference to itself

setEnv (*pEnv*)

Sets taskgroup environment dict with the given param.

Parameters **pEnv** – the new environment

updateTags (*pTags*)

Updates the tag dicitonnary for this element.

Parameters **pTags** – a dict of tags (with existing or new values)

3.4 Task

class puliclient.**Task**

A node of the graph that can contain commands i.e. processes that will be executed on a rendernode.

__init__ (*name*, *arguments*, *runner='puliclient.jobs.DefaultCommandRunner'*, *decomposer='puliclient.jobs.DefaultTaskDecomposer'*, *runnerPackages=None*, *watcherPackages=None*, *dependencies={}*, *maxRN=0*, *priority=0*, *dispatchKey=0*, *environment={}*, *validator='0'*, *minNbCores=1*, *maxNbCores=0*, *ramUse=0*, *requirements={}*, *lic=''*, *tags={}*, *timer=None*, *maxAttempt=1*)

A task contains one or more command to be executed on the render farm. The parameters that will be used to create commands (the process of decomposing) are the following: arguments, runner and decomposer Other params are mainly used on the Task, to handle matching and dispatching on the server.

Parameters

- **str** (*lic*) – A simple text identifier
- **dict** (*tags*) – a dictionnary of arguments for the command
- **str** – class that will be responsible for the job execution
- **decomposer** – class responsible to create commands relative to this task
- **dependencies** – a list of nodes and result status from which the current task depends on
- **int** (*timer*) – the maximum number of workers to assign to this task
- **priority** – deprecated
- **int** – indicate the priority for this task
- **dict** – A set of env values
- **validator** – deprecated
- **minNbCores** – deprecated
- **maxNbCores** – deprecated

- **int** – the amount of RAM in megabytes needed on a render node to be assigned with a command of this task
- **requirements** – deprecated
- **str** – a flag indicating one or several licence token to reserve for each command
- **dict** – user defined values
- **int** – a date (as a timestamp) to wait before assigning commands of the current task

addCommand(*name*, *arguments*, *runnerPackages=None*, *watcherPackages=None*)

Manually add a command on the current node.

Parameters

- **name** – a custom name for this command
- **arguments** – dict of arguments for the specific command

decompose()

Call the task "decomposer", a utility class that will generate one | or several command regarding decomposing method.

For instance given a start and end attributes, we will created a | sequence of command.

if the task has no decomposer defined (when user manually add | commands), simply "visit" the task

Returns a ref to itself

dependsOn(*task, statusList=[3]*)

Create a dependency constraint between the current node and the given task at a particular status.

Parameters

- **task** – a task to dependsOn
- **statusList** – a list of statuses to be reached (any of it) to validate the dependency

updateTags(*pTags*)

Updates the tag dicitonnary for this element.

Parameters **pTags** – a dict of tags (with existing or new values)

3.5 Command

class puliclient.**Command**

The lowest level of execution of a graph. A command is basically a process
to instanciate on a worker node.
It will use the "runner" class of its parent task for its execution.

It consists of:
- a description
- a ref to its parent task
- a dict of arguments to use for execution (of the task's runner)

A default runner can handle the following arguments:

- cmd: a string indicating a command line to execute

- timeout: a positive integer indicating the max number of second that the command can run before being interrupted

CHAPTER 4

TODO - PULI Dev guide

4.1 Source tree presentation

4.2 Persistence and database

4.3 Assignment

CHAPTER 5

IN PROGRESS - Module reference

5.1 Dispatcher

5.2 Db

5.3 LicenseManager

Created on 25 nov. 2009

class `octopus.dispatcher.licenses.licensemanager.`**`LicenseManager`**

`releaseLicenseForRenderNode`(*licenseName*, *renderNode*)

LicenseName

RenderNode render node object expected

`stats`()

Get useful information on licenses declared on the server.

Returns a list of dict, each of them being a license description like { 'name': 'shave', 'total': 70, 'used': 0, 'rns': [] }

CHAPTER 6

Indices and tables

- *genindex*
- *modindex*
- *search*

o

p

Symbols

A

C

D

E

G

L

O

P

R

S

T

U