RADICAL-Pilot Documentation

Release 0.50.21

The RADICAL Group at Rutgers University

Dec 21, 2018

Contents

RADICAL-Pilot (RP) is a Pilot Job system written in Python. It allows a user to run large numbers of computational tasks (called `ComputeUnits`) concurrently on one or more remote `ComputePilots` that RADICAL-Pilot can start transparently on a multitude of different distributed resources, like HPC clusters and Clouds.

In this model, a part (slice) of a resource is acquired by a user's application so that the application can directly schedule `ComputeUnits` into that resource slice, rather than going through the system's job scheduler. In many cases, this can drastically shorten overall exeuction time as the individual `ComputeUnits` don't have to wait in the system's scheduler queue but can execute directly on the `ComputePilots`.

`ComputeUnits` can be sequential, multi-threaded (e.g. OpenMP), parallel process (e.g. MPI) executables, Hadoop or Spark applications.

RADICAL-Pilot is not a static system, but it rather provides the user with a programming library ("Pilot-API") that provides abstractions for resource access and task management. With this library, the user can develop everything from simple "submission scripts" to arbitrarily complex applications, higher- level services and tools.

Links

- repository: https://github.com/radical-cybertools/radical.pilot
- user list: https://groups.google.com/d/forum/radical-pilot-users
- developer list: https://groups.google.com/d/forum/radical-pilot-devel

CHAPTER 1

Contents:

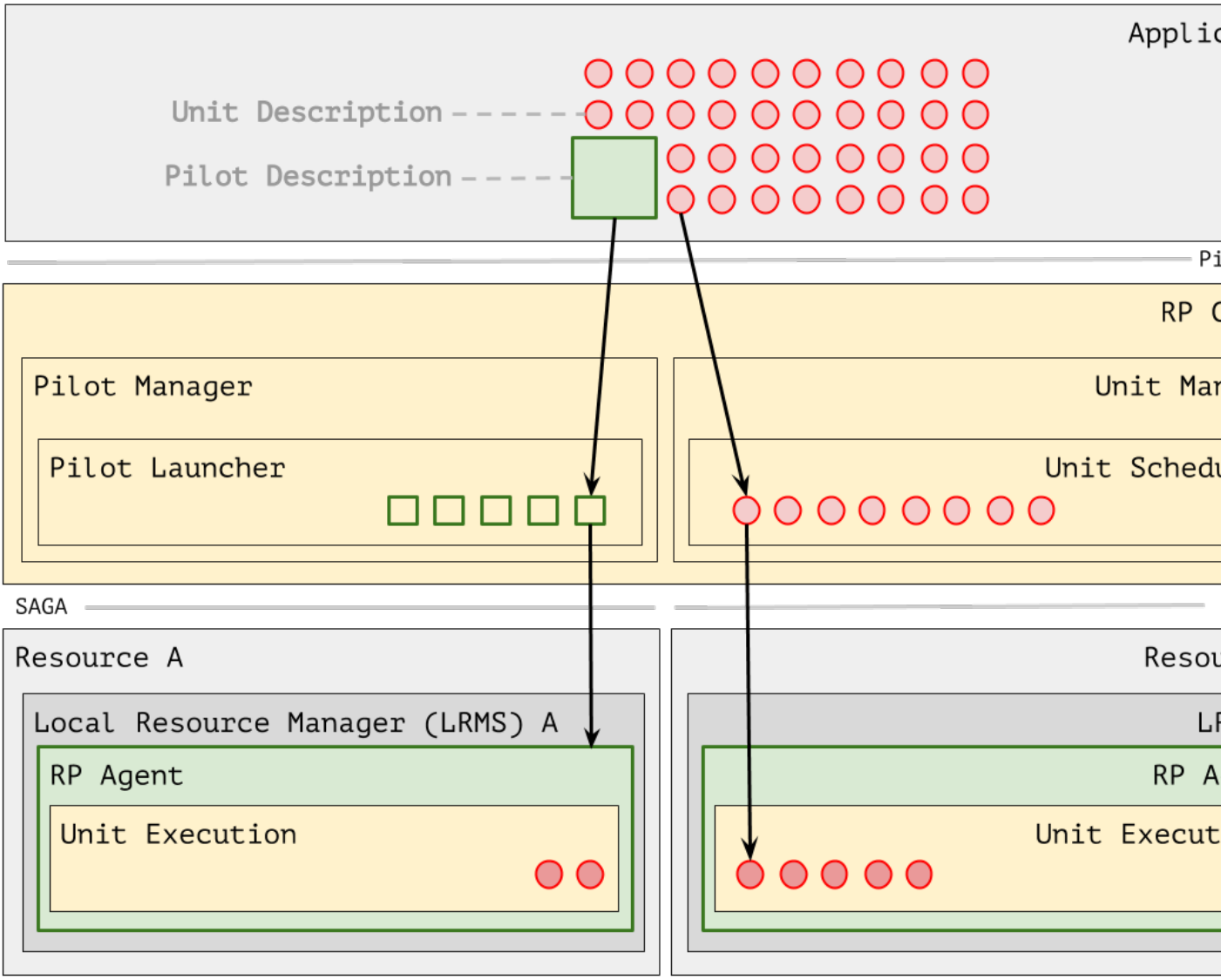
1.1 Introduction

RADICAL-Pilot (RP) is a Pilot Job system written in Python. It allows a user to run large numbers of computational tasks (called `ComputeUnits`) concurrently on one or more remote `ComputePilots` that RADICAL-Pilot can start transparently on a multitude of different distributed resources, like HPC clusters and Clouds.

In this model, a part (slice) of a resource is acquired by a user's application so that the application can directly schedule `ComputeUnits` into that resource slice, rather than going through the system's job scheduler. In many cases, this can drastically shorten overall exeuction time as the individual `ComputeUnits` don't have to wait in the system's scheduler queue but can execute directly on the `ComputePilots`.

`ComputeUnits` are often single-core / multi-threaded executables, but RADICAL-Pilot also supports execution of parallel executables, for example based on MPI, OpenMP, YARN/HADOOP and Spark.

RADICAL-Pilot is not a static system, but it rather provides the user with a programming library ("Pilot-API") that provides abstractions for resource access and task management. With this library, the user can develop everything from simple "submission scripts" to arbitrarily complex applications, higher- level services and tools.



The RP architecture overview image above shows the main components or RP, and their functional relationships. The RP system will interpret pilot descriptions and submit the respective pilot instances on the target resources. It will then accept unit descriptions and submit those for execution onto the earlier created pilots. The Chapter *RADICAL-Pilot Overview* (page 4) will discuss those concepts in some more detail, before then turning to *Installation* (page 9) which will describe how RP is deployed and configured, so that the reader can follow the upcoming examples.

1.2 RADICAL-Pilot - Overview

This section provides a conceptual overview about RADICAL-Pilot (RP). You will learn what problems RP aims to solve for you. You will also be introduced to some vocabulary, and the overall RP architecture and operation.

We will keep the information on a very general level, and will avoid any details which will not contribute to the user experience. Having said that, feel free to skip ahead to the *User Guide* (page 12) if you are more interested in directly diving into the thick of using RP!

1.2.1 What Problems does RP solve?

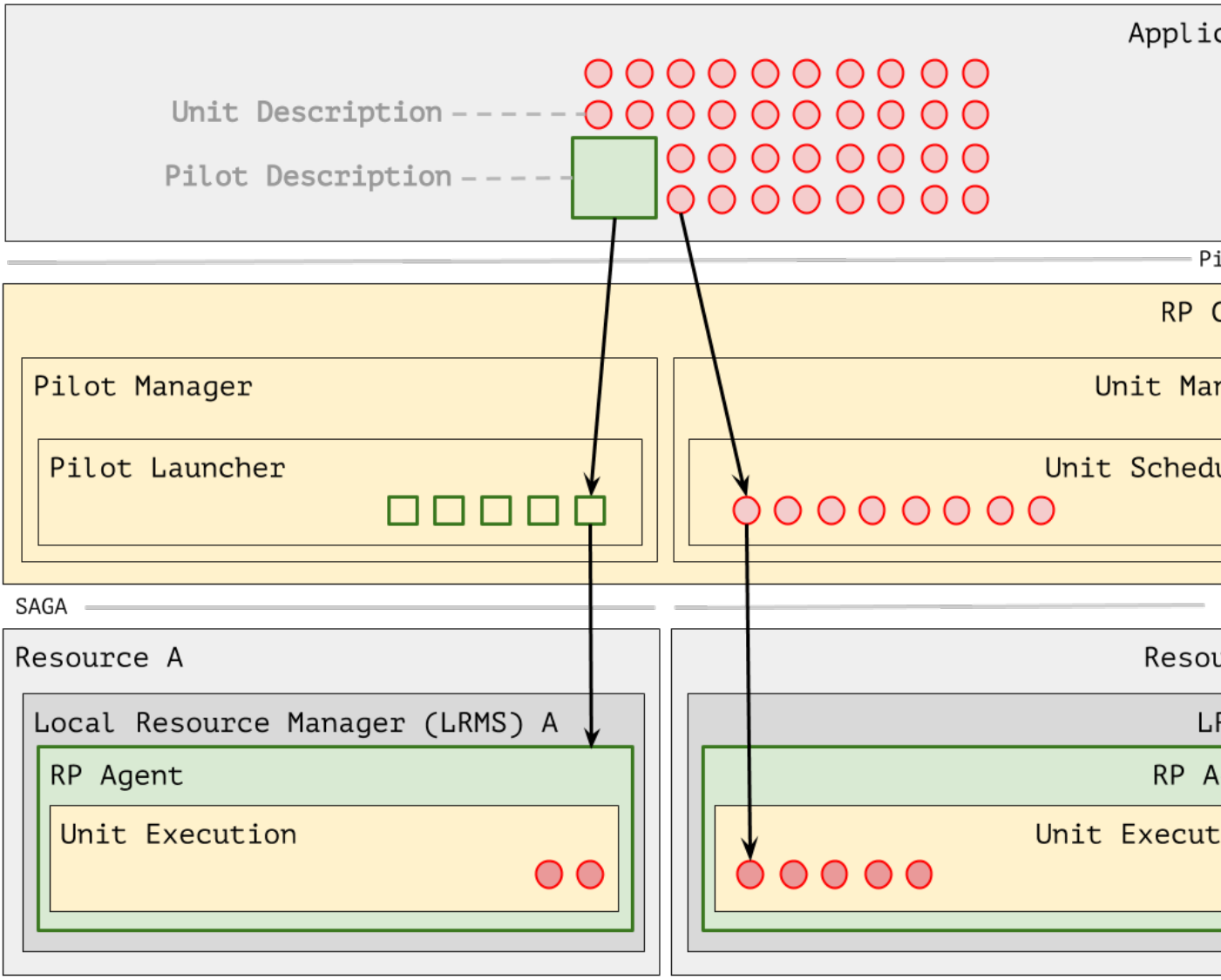
RP attempts to support in running applications on distributed resources, and focuses on two aspects:

- abstract the heterogeneity of distributed resources, so that running applications is uniform across them, from a users perspective;
- support the efficient execution of large numbers of concurrent or sequential application instances.

1.2.2 What is a Pilot?

The `Pilot` in RADICAL-Pilot stands for a job-container like construct which manages a part (*slice*) of a remote resource on the user's (or application's) behalf, and which executes sequences of `ComputeUnits` on that resource slice.

RP applications will, in general, first define a set of such pilots, ie. the set of target resources, the size of the resource slice to manage, etc), and then submit those pilots to the resources. Once the pilots are defined, the application can send them `ComputeUnits` (see below) for execution.



1.2.3 What is a Compute Unit (CU)?

An RP ComputeUnit (CU, or 'unit') represents a self-contained, executable part of the application's workload. A CU is described by the following attributes (for details, check out the *API documentation* (page 52)):

- *executable* : the name of the executable to be run on the target machines
- *arguments* : a list of argument strings to be passed to the executable
- *environment* : a dictionary of environment variable/value pairs to be set before unit execution
- *input_staging* : a set of staging directives for input data
- *output_staging*: a set of staging directives for output data

1.2.4 How about data?

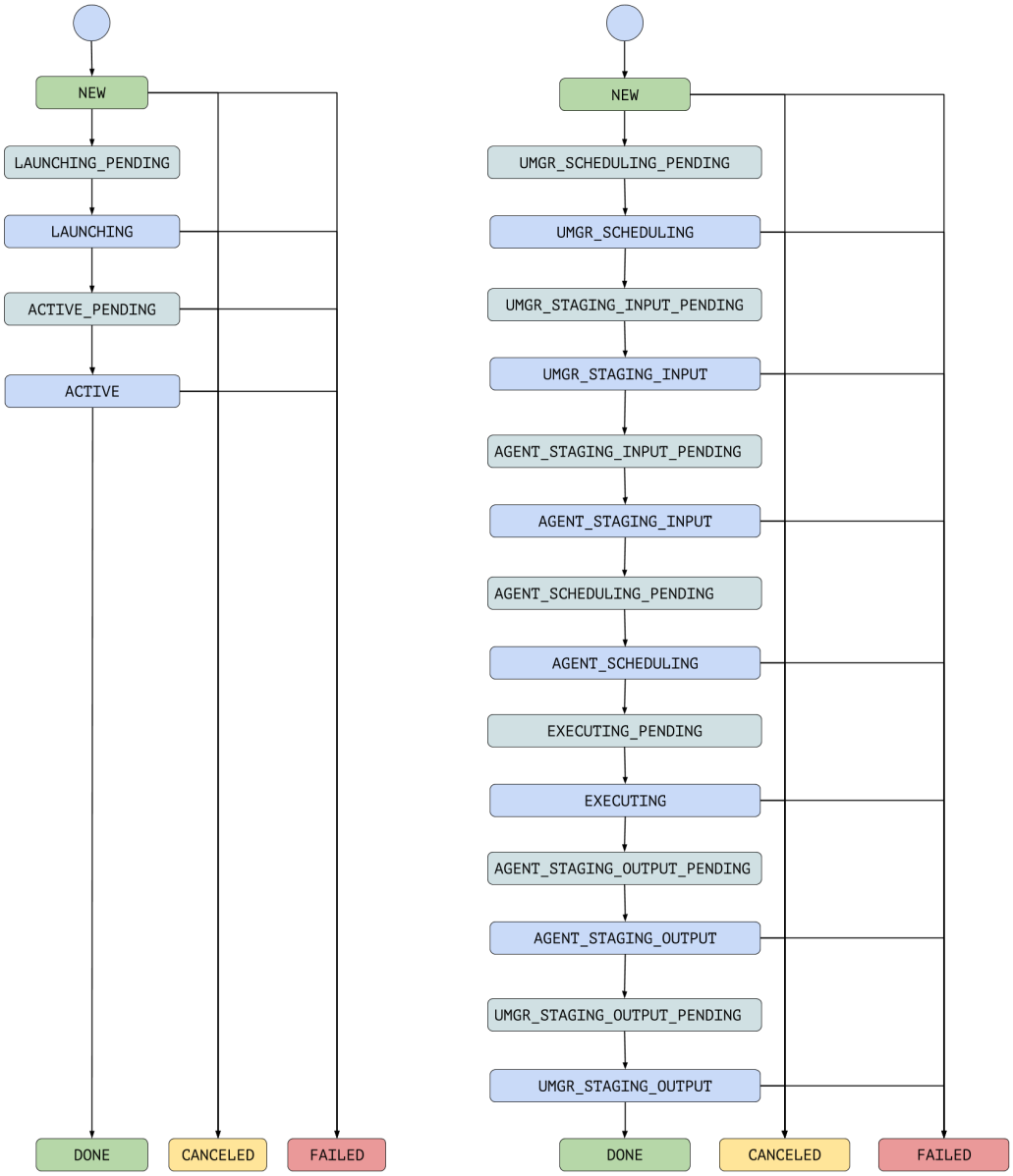
Data management is important for executing CUs, both in providing input data, and staging/sharing output data. RP has different means to handle data, and they are specifically covered in sections *in* (page 26) *the* (page 28) *UserGuide* (page 30).

1.2.5 Why do I need a MongoDB to run RP?

The RP application uses a MongoDB database to communicate with the pilots it created: upon startup, the pilots will connect to the database and look for CUs to execute. Similarly, pilots will push information into the database, such as about units which completed execution.

1.2.6 How do I know what goes on in the pilot? With my CUs?

There are many aspects to that question. First, pilots and units progress according to well defined state models:



`pilot.state` and `unit.state` will always report the current state of the entities.

Callbacks can be registered for notifications on unit and pilot state changes.

Setting `RADICAL_PILOT_VERBOSE=INFO` will turn on logging, which provides very detailed information about RP's inner functionality. Pilots running on target resources also create log files (those should only be useful for debugging purposes).

1.2.7 What about logging and profiling?

RP supports logging to the terminal and to files. Also, profiles can be written during runtime. You can set the following variables:

RADICAL_PILOT_VERBOSE=DEBUG RADICAL_PILOT_LOG_TGT=/tmp/rp.log RADICAL_PILOT_PROFILE=True

The defined verbosity levels are the same as defined by Python's logging module

1.3 Installation

1.3.1 Requirements

RADICAL-Pilot requires the following packages:

- Python >= 2.7 (including development headers)
- virtualenv >= 1.11
- pip == 1.4.1

or * Anaconda Python 2.7

If you plan to use RADICAL-Pilot on remote machines, you would also require to setup a password-less ssh login to the particular machine. (help)

All dependencies are installed automatically by the installer. Besides that, RADICAL-Pilot needs access to a MongoDB database that is reachable from the internet. User groups within the same institution or project usually share a single MongoDB instance. MongoDB is standard software and available in most Linux distributions. At the end of this section, we provide brief instructions how to set up a MongoDB server and discuss some advanced topics, like SSL support and authentication to increased the security of RADICAL-Pilot.

1.3.2 Installation

RADICAL-Pilot is distributed via PyPi and Conda-Forge. To install RADICAL-Pilot to a virtual environment do:

via PyPi

```
virtualenv --system-site-packages $HOME/ve
source $HOME/ve/bin/activate
pip install radical.pilot
```

via Conda-Forge

```
conda create -n ve -y python=2.7
source activate ve
conda install radical.pilot -c conda-forge
```

For a quick sanity check, to make sure that the the packages have been installed properly, run:

```
$ radicalpilot-version
0.50.8
```

The exact output will obviously depend on the exact version of RP which got installed.

** Installation is complete !**

1.3.3 Preparing the Environment

MongoDB Service

RP requires access to a MongoDB server. The MongoDB server is used to store and retrieve operational data during the execution of an application using RADICAL-Pilot. The MongoDB server must be reachable from **both**, the host that runs the RP application and the target resource which runs the pilots.

> **Warning:** If you want to run your application on your laptop or private workstation, but run your MD tasks on a remote HPC cluster, installing MongoDB on your laptop or workstation won't work. Your laptop or workstations usually does not have a public IP address and is hidden behind a masked and firewalled home or office network. This means that the components running on the HPC cluster will not be able to access the MongoDB server.

Any MongoDB installation should work out, as long as RP is allowed to create databases and collections (which is the default user setting in MongoDB).

The MongoDB location is communicated to RP via the environment variable `RADICAL_PILOT_DBURL`. The value will have the form

```
export RADICAL_PILOT_DBURL="mongodb://user:pass@host:port/dbname"
export RADICAL_PILOT_DBURL="mongodb://host:port/dbname"
```

Many MongoDB instances are by default unsecured, and thus do not require the `user:pass@` part of the URL. For production runs, and for runs on large secured resources, we strongly recommend the usage of a secured MongoDB instance!

The `dbname` component in the database url can be any valid MongoDB database name (ie. it musy not contain dots).RP will not create that DB on the fly and requires the DB to be setup prior to creating the session object. But RP will create collections in that DB on its own, named after RP session IDs.

A MongoDB server can support more than one user. In an environment where multiple users use RP applications, a single MongoDB server for all users / hosts is usually sufficient. We recommend the use of separate databases per user though, so please set the `dbname` to something like `db_joe_doe`.

Install your own MongoDB

Once you have identified a host that can serve as the new home for MongoDB, installation is straight forward. You can either install the MongoDB server package that is provided by most Linux distributions, or follow the installation instructions on the MongoDB website:

- http://docs.mongodb.org/manual/installation/

MongoDB-as-a-Service

There are multiple commercial providers of hosted MongoDB services, some of them offering free usage tiers. We have had some good experience with the following:

- https://mongolab.com/

Setup SSH Access to Target Resources

An easy way to setup SSH Access to multiple remote machines is to create a file `~/.ssh/config`. Suppose the url used to access a specific machine is `foo@machine.example.com`. You can create an entry in this config file as follows:

```
# contents of $HOME/.ssh/config
Host mach1
    HostName machine.example.com
    User foo
```

Now you can login to the machine by using `ssh mach1`. Please make also sure that your ssh keys are registered on the target resources – while RP can in principle handle password based login, the repeated prompts for passwords makes RP applications very difficult to use.

Source: http://nerderati.com/2011/03/17/simplify-your-life-with-an-ssh-config-file/

1.3.4 Troubleshooting

Missing virtualenv

This should return the version of the RADICAL-Pilot installation, e.g., `0.X.Y`.

If virtualenv **is not** installed on your system, you can try the following.

```
wget --no-check-certificate https://pypi.python.org/packages/source/v/virtualenv/
↪virtualenv-1.9.tar.gz
tar xzf virtualenv-1.9.tar.gz

python virtualenv-1.9/virtualenv.py $HOME/ve
source $HOME/ve/bin/activate
```

Installation Problems

Many installation problems boil down to one of two causes: an Anaconda based Python distribution, or an incompatible version of pip/setuptools.

Many recent systems, specifically in the academic community, install Python in its incarnation as Anaconda Distribution. RP is not yet able to function in that environment. While support of Anaconda is planned in the near future, you will have to revert to a 'normal' Python distribution to use RP.

Python supports a large variety of module deployment paths: `easy_install`, `setuptools` and `pip` being the most prominent ones for non-compilable modules. RP only supports `pip`, and even for pip we do not attempt to keep up with its vivid evolution. If you encounter pip errors, please downgrade pip to version `1.4.1`, via

```
$ pip install --upgrade pip==1.4.1
```

If you continue to encounter problems, please also fix the version of setuptools to `0.6c11` via

```
$ pip install --upgrade setuptools==0.6c11
```

Note: RADICAL-Pilot can be installed under Anaconda, although that mode is not tested as thoroughly compared to installation under non-Anaconda Python.

Mailing Lists

If you encounter any errors, please do not hesitate to contact us via the mailing list:

- https://groups.google.com/d/forum/radical-pilot-users

We also appreciate issues and bug reports via our public github tracker:

- https://github.com/radical-cybertools/radical.pilot/issues

1.4 User Guide

This chapter will guide the reader through the most commonly used features of RADICAL-Pilot (RP). We will start with a basic RP example, and then discuss individual RP features which augment the basic example towards a more complete set of applications.

1.4.1 Getting Started

In this section we will walk you through the basics of using RP. After you have worked through this chapter, you will understand how to launch a local `ComputePilot` and use a `UnitManager` to schedule and run `ComputeUnits` (tasks) on it.

Note: The reader is assumed to be familiar with the general RP concepts as described in *RADICAL-Pilot - Overview* (page 4) for reference.

Note: This chapter assumes that you have successfully installed RADICAL-Pilot, and also configured access to the resources you intent to use for the examples (see chapter *Installation* (page 9)).

Note: We colloquially refer to `ComputePilot` as *pilot*, and to `ComputeUnit` as *unit*.

You can download the basic `00_getting_started.py`. The text below will explain the most important code sections, and at the end shows the expected output from the execution of the example. Please look carefully at the code comments as they explain some aspects of the code which are not explicitly covered in the text below.

Loading the RP Module, Follow the Application Execution

In order to use RADICAL-Pilot, you need to import the `radical.pilot` module (we use the *rp* abbreviation for the module name) in your Python script or application:

```
import radical.pilot as rp
```

All example scripts used in this user guide use the `LogReporter` facility (of RADICAL-Utils) to print runtime and progress information. You can control that output with the `RADICAL_PILOT_VERBOSE` variable, which can be set to the normal Python logging levels, and to the value `REPORT` to obtain well formatted output. We assume the `REPORT` setting to be used when referencing any output in this chapter.

```
os.environ['RADICAL_PILOT_VERBOSE'] = 'REPORT'

import radical.pilot as rp
import radical.utils as ru

report = ru.LogReporter(name='radical.pilot')
report.title('Getting Started (RP version %s)' % rp.version)
```

Creating a Session

A *radical.pilot.Session* (page 40) is the root object for all other objects in RADICAL- Pilot. *radical.pilot.PilotManager* (page 43) and *radical.pilot.UnitManager* (page 49) instances are always at-

tached to a Session, and their lifetime is controlled by the session.

A Session also encapsulates the connection(s) to a backend MongoDB server which facilitates the communication between the RP application and the remote pilot jobs. More information about how RADICAL-Pilot uses MongoDB can be found in the *RADICAL-Pilot - Overview* (page 4) section.

To create a new Session, the only thing you need to provide is the URL of a MongoDB server. If no MongoDB URL is specified on session creation, RP attempts to use the value specified via the `RADICAL_PILOT_DBURL` environment variable.

```
os.environ['RADICAL_PILOT_DBURL'] = 'mongodb://db.host.net:27017/<db_name>'

session = rp.Session()
```

Warning: Always call *radical.pilot.Session.close()* (page 41) before your application terminates. This will terminate all lingering pilots and cleans out the database entries of the session.

Creating ComputePilots

A *radical.pilot.ComputePilot* (page 46) is responsible for `ComputeUnit` execution. Pilots can be launched either locally or remotely, and they can manage a single node or a large number of nodes on a cluster.

Pilots are created via a *radical.pilot.PilotManager* (page 43), by passing a *radical.pilot.ComputePilotDescription* (page 45). The most important elements of the `ComputePilotDescription` are

- *resource*: a label which specifies the target resource to run the pilot on, ie. the location of the pilot;
- *cores* : the number of CPU cores the pilot is expected to manage, ie. the size of the pilot;
- *runtime* : the numbers of minutes the pilot is expected to be active, ie. the runtime of the pilot.

Depending on the specific target resource and use case, other properties need to be specified. In our user guide examples, we use a separate *config.json* file to store a number of properties per resource label, to simplify the example code. The examples themselves then accept one or more resource labels, and create the pilots on those resources:

```
# use the resource specified as argument, fall back to localhost
try   : resource = sys.argv[1]
except: resource = 'local.localhost'

# create a pilot manage in the session
pmgr = rp.PilotManager(session=session)

# define an [n]-core local pilot that runs for [x] minutes
pdesc = rp.ComputePilotDescription({
        'resource'      : resource,
        'cores'         : 64,  # pilot size
        'runtime'       : 10,  # pilot runtime (min)
        'project'       : config[resource]['project'],
        'queue'         : config[resource]['queue'],
        'access_schema' : config[resource]['schema']
        }

# submit the pilot for launching
pilot = pmgr.submit_pilots(pdesc)
```

For a list of available resource labels, see chapter_resources (not all of those resources are configured for the userguide examples). For further details on the pilot description, please check the *API Documentation* (page 45).

Warning: Note that the submitted pilot agent **will not terminate** when your Python scripts finishes. Pilot agents terminate only after they have reached their `runtime` limit, are killed by the target system, or if you explicitly cancel them via `radical.pilot.Pilot.cancel()`, *radical.pilot.PilotManager.cancel_pilots()* (page 45), or `radical.pilot.Session.close(terminate=True)()`.

Submitting ComputeUnits

After you have launched a pilot, you can now generate *radical.pilot.ComputeUnit* (page 55) objects for the pilot to execute. You can think of a `ComputeUnit` as something very similar to an operating system process that consists of an `executable`, a list of `arguments`, and an `environment` along with some runtime requirements.

Analogous to pilots, a units is described via a *radical.pilot.ComputeUnitDescription* (page 52) object. The mandatory properties that you need to define are:

- `executable` - the executable to launch
- `cores` - the number of cores required by the executable

Our basic example creates 128 units which each run */bin/date*:

```
n    = 128   # number of units to run
cuds = list()
for i in range(0, n):
    # create a new CU description, and fill it.
    cud = rp.ComputeUnitDescription()
    cud.executable = '/bin/date'
    cuds.append(cud)
```

Units are executed by pilots. The *:class:radical.pilot.UnitManager* class is responsible for routing those units from the application to the available pilots. The `UnitManager` accepts `ComputeUnitDescriptions` as we created above and assigns them, according to some scheduling algorithm, to the set of available pilots for execution (pilots are made available to a `UnitManager` via the `add_pilot` call):

```
# create a unit manager, submit units, and wait for their completion
umgr = rp.UnitManager(session=session)
umgr.add_pilots(pilot)
umgr.submit_units(cuds)
umgr.wait_units()
```

Running the Example

Note: Remember to set *RADICAL_PILOT_DBURL* in you environment (see chapter *Installation* (page 9)).

Running the example will result in an output similar to the one shown below:

```
================================================================================
 Getting Started (RP version v0.36.RC1)
================================================================================

create session rp.session.cameo.merzky.016719.0000                          ok
read config                                                                 ok

--------------------------------------------------------------------------------
submit pilots

create pilot manager                                                        ok
create pilot description                                                     \
create pilot description [local.localhost:64]                               ok
                                                                            ok
submit 1 pilot(s) .                                                         ok

--------------------------------------------------------------------------------
submit units

create unit manager                                                         ok
add 1 pilot(s)                                                              ok
create 128 unit description(s)
        ................................................................................
        ........................................................                ok
submit 128 unit(s)
        ................................................................................
        ........................................................                ok

--------------------------------------------------------------------------------
gather results

wait for 128 unit(s)
        ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++|
        ++++++++++++++++++++++++++++++++++++++++++++++++++++++++                  ok

--------------------------------------------------------------------------------
finalize

closing session rp.session.cameo.merzky.016719.0000                          \
close pilot manager                                                          \
wait for 1 pilot(s) *                                                       ok
                                                                            ok
close unit manager                                                          ok
                                                                            ok

--------------------------------------------------------------------------------
```

The runtime can vary significantly, and typically the first run on any resource will be longest. This is because the first time RP is used on a new resource for a specific user, it will set up a Python virtualenv for the pilot to use. Subsequent runs may update that virtualenv, or may install additional components as needed, but that should take less time than its creation. So please allow for a couple of minutes on the first execution (depending on your network connectivity, the connectivity of the target resource, and the location of the MongoDB service).

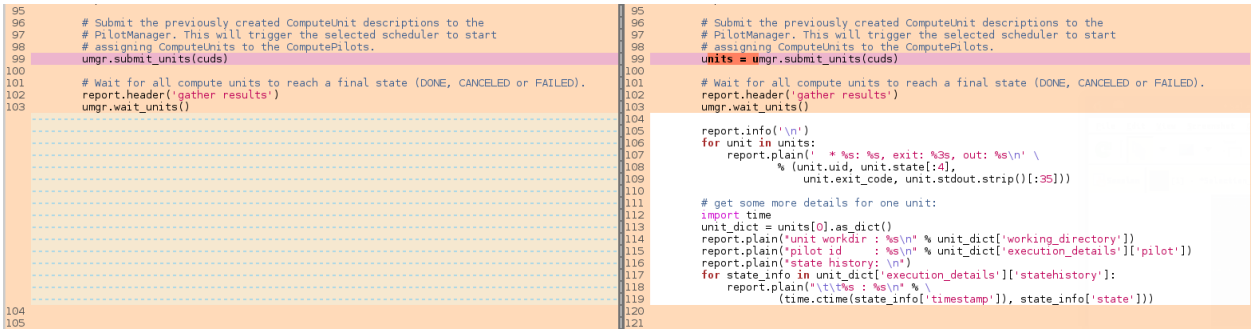
What's Next?

The next user guide section (*Obtaining Unit Details* (page 16)) will describe how an application can inspect completed units for more detailed information, such as exit codes and stdout/stderr.

1.4.2 Obtaining Unit Details

The *previous chapter* (page 12) discussed the basic RP features, how to submit a pilot, and how to submit ComputeUnits to that pilot for execution. We will here show how an application can, after the units complete, inspect the details of that execution.

You can download the script `01_unit_details.py`, which has the following diff to the basic example:



```
95
96      # Submit the previously created ComputeUnit descriptions to the
97      # PilotManager. This will trigger the selected scheduler to start
98      # assigning ComputeUnits to the ComputePilots.
99      umgr.submit_units(cuds)
100
101     # Wait for all compute units to reach a final state (DONE, CANCELED or FAILED).
102     report.header('gather results')
103     umgr.wait_units()
104
105
```

```
95
96      # Submit the previously created ComputeUnit descriptions to the
97      # PilotManager. This will trigger the selected scheduler to start
98      # assigning ComputeUnits to the ComputePilots.
99      units = umgr.submit_units(cuds)
100
101     # Wait for all compute units to reach a final state (DONE, CANCELED or FAILED).
102     report.header('gather results')
103     umgr.wait_units()
104
105     report.info('\n')
106     for unit in units:
107         report.plain('  * %s: %s, exit: %3s, out: %s\n' \
108                 % (unit.uid, unit.state[:4],
109                     unit.exit_code, unit.stdout.strip()[:35]))
110
111     # get some more details for one unit:
112     import time
113     unit_dict = units[0].as_dict()
114     report.plain("unit workdir : %s\n" % unit_dict['working_directory'])
115     report.plain("pilot id     : %s\n" % unit_dict['execution_details']['pilot'])
116     report.plain("state history: \n")
117     for state_info in unit_dict['execution_details']['statehistory']:
118         report.plain("\t\t%s : %s\n" % \
119                 (time.ctime(state_info['timestamp']), state_info['state']))
120
121
```

You'll notice that we capture the return value of *submit_units()* in line 99, which is in fact a list of ComputeUnit instances. We use those instances for inspection later on, after we waited for completion. Inspection is also available earlier, but may then, naturally, yield incomplete results. A unit will *always* have a state though, according to the state model discussed in *RADICAL-Pilot - Overview* (page 4).

The code block below contains what most applications are interested is: unit state, exit code, and standard output (we'll see *later* (page 18) that stderr is handled equivalently):

```
report.plain('  * %s: %s, exit: %3s, out: %s\n' \
        % (unit.uid, unit.state[:4],
            unit.exit_code, unit.stdout.strip()[:35]))
```

Note: The reporting of standard output in this manner is a convenience method, and cannot replace proper output file staging: the resulting string will be shortened on very long outputs (longer than 1kB by default), and it may contain information from RP which are not strictly part of the application stdout messages. The proper staging of output file will be discussed in a :ref:later <chapter_user_guide_06>' example.

Running the Example

Running the example will result in an output similar to the one shown below:

```
================================================================================
 Getting Started (RP version v0.36.RC1)
================================================================================

create session rp.session.cameo.merzky.016720.0006                          ok
read config                                                                 ok

--------------------------------------------------------------------------------
submit pilots

create pilot manager                                                        ok
create pilot description                                                     \
create pilot description [local.localhost:64]                               ok
                                                                            ok
submit 1 pilot(s) .                                                         ok

--------------------------------------------------------------------------------
submit units

create unit manager                                                         ok
add 1 pilot(s)                                                              ok
create 128 unit description(s)
       ..................................................................
       ..........................................................           ok
submit 128 unit(s)
       ..................................................................
       ..........................................................           ok

--------------------------------------------------------------------------------
gather results

wait for 128 unit(s)
       +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++|
       ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++           ok

  * unit.000000: Done, exit:   0, out: Mon Oct 12 12:39:06 CEST 2015
  * unit.000001: Done, exit:   0, out: Mon Oct 12 12:38:46 CEST 2015
  * unit.000002: Done, exit:   0, out: Mon Oct 12 12:38:46 CEST 2015
  * [...]
  * unit.000125: Done, exit:   0, out: Mon Oct 12 12:39:08 CEST 2015
  * unit.000126: Done, exit:   0, out: Mon Oct 12 12:39:08 CEST 2015
  * unit.000127: Done, exit:   0, out: Mon Oct 12 12:39:08 CEST 2015
unit workdir : file://localhost/home/merzky/radical.pilot.sandbox/ \
               rp.session.cameo.merzky.016720.0006-pilot.0000//unit.000000
pilot id     : pilot.0000
state history:
               Mon Oct 12 12:38:28 2015 : Scheduling
               Mon Oct 12 12:39:05 2015 : StagingInput
               Mon Oct 12 12:39:05 2015 : AgentStagingInputPending
               Mon Oct 12 12:39:06 2015 : AgentStagingInput
               Mon Oct 12 12:39:06 2015 : AllocatingPending
               Mon Oct 12 12:39:06 2015 : Allocating
               Mon Oct 12 12:39:06 2015 : ExecutingPending
               Mon Oct 12 12:39:06 2015 : Executing
               Mon Oct 12 12:39:06 2015 : AgentStagingOutputPending
               Mon Oct 12 12:39:06 2015 : AgentStagingOutput
               Mon Oct 12 12:39:06 2015 : PendingOutputStaging
               Mon Oct 12 12:39:18 2015 : StagingOutput
               Mon Oct 12 12:39:18 2015 : Done

--------------------------------------------------------------------------------
finalize

closing session rp.session.cameo.merzky.016720.0006                          \
close pilot manager                                                          \
wait for 1 pilot(s) *                                                       ok
                                                                            ok
```

What's Next?

The next user guide section (*Handle Failing Units* (page 18)) will describe how failed units can be differentiated from successful ones – although the avid reader will already have an intuition on how that is done.

1.4.3 Handle Failing Units

All applications can fail, often for reasons out of control of the user. A ComputeUnit is no different, it can fail as well. Many non-trivial application will need to have a way to handle failing units – detecting the failure is the first and necessary step to do so, and RP makes that part easy: RP's unit state model defines that a failing unit will immediately go into *FAILED* state, and that state information is available as *unit.state* property.

The unit also has the *unit.stderr* property available for further inspection into causes of the failure – that will only be available though if the unit did reach the *EXECUTING* state in the first place. In other cases, the application can inspect the *unit.as_dict()['execution_details']['log']* array of timed log messages, similar to the *state_history* array discussed *before* (page 16).

You can download the script `02_failing_units.py`, which demonstrates inspection for failed units. It has the following diff to the previous example:

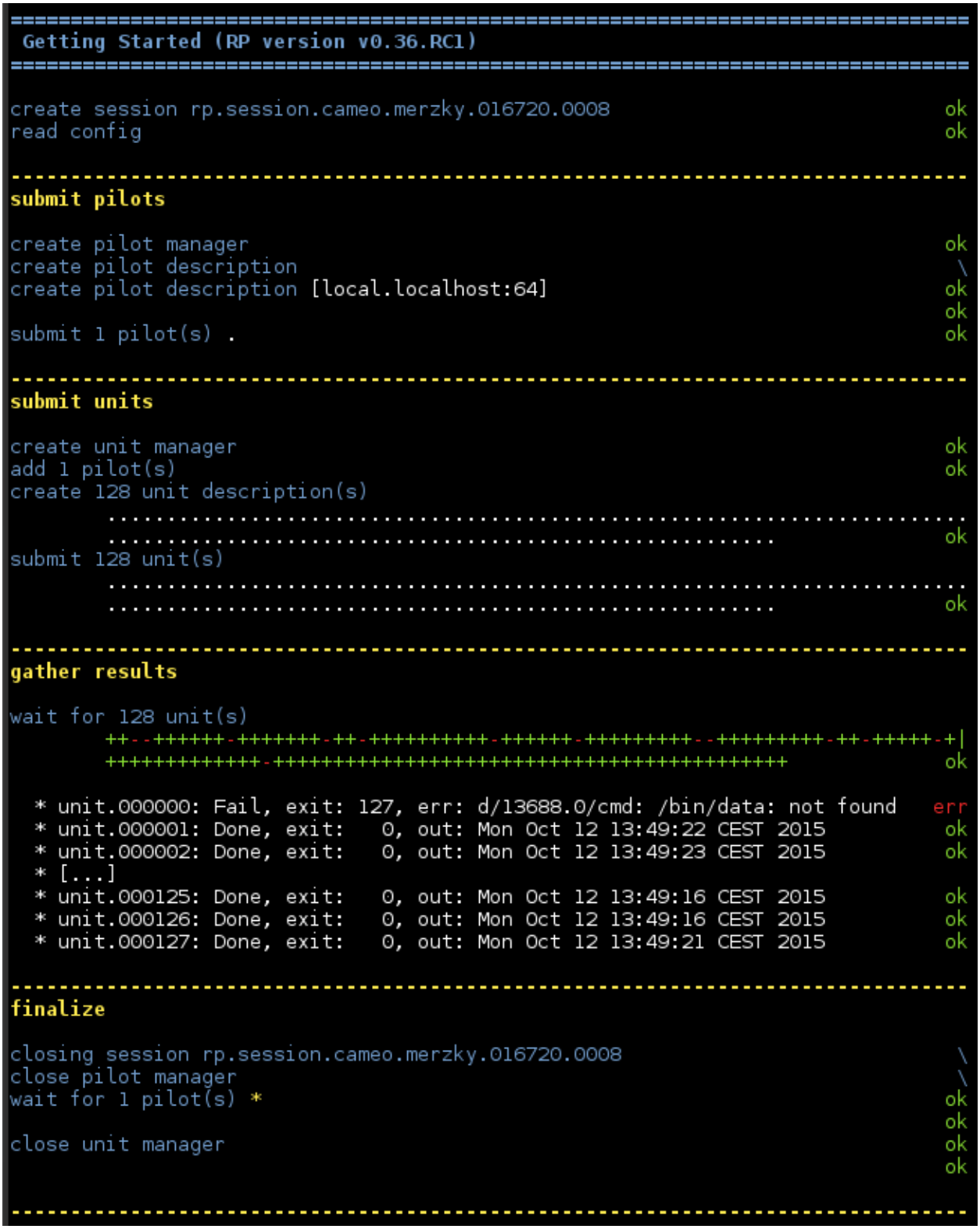
```
90          cud = rp.ComputeUnitDescription()
91          cud.executable = '/bin/date'
92          cuds.append(cud)
93          report.progress()
94      report.ok('>>ok\n')
95
96      # Submit the previously created ComputeUnit descriptions to the
97      # PilotManager. This will trigger the selected scheduler to start
98      # assigning ComputeUnits to the ComputePilots.
99      units = umgr.submit_units(cuds)
100
101     # Wait for all compute units to reach a final state (DONE, CANCELED or FAILED).
102     report.header('gather results')
103     umgr.wait_units()
104
105     report.info('\n')
106     for unit in units:
107         report.plain('  * %s: %s, exit: %3s, out: %s\n' \
108                % (unit.uid, unit.state[:4],
109                   unit.exit_code, unit.stdout.strip()[:35]))
110
111     # get some more details for one unit:
112     import time
113     unit_dict = units[0].as_dict()
114     report.plain("unit workdir : %s\n" % unit_dict['working_directory'])
115     report.plain("pilot id     : %s\n" % unit_dict['execution_details']['pilot'])
116     report.plain("state history: \n")
117     for state_info in unit_dict['execution_details']['statehistory']:
118         report.plain("\t\t%s : %s\n" % \
119                (time.ctime(state_info['timestamp']), state_info['state']))
120
```

```
90          cud = rp.ComputeUnitDescription()
91
92          # trigger an error now and then
93          if not i % 10: cud.executable = '/bin/data' # does not exist
94          else         : cud.executable = '/bin/date'
95
96          cuds.append(cud)
97          report.progress()
98      report.ok('>>ok\n')
99
100     # Submit the previously created ComputeUnit descriptions to the
101     # PilotManager. This will trigger the selected scheduler to start
102     # assigning ComputeUnits to the ComputePilots.
103     units = umgr.submit_units(cuds)
104
105     # Wait for all compute units to reach a final state (DONE, CANCELED or FAILED).
106     report.header('gather results')
107     umgr.wait_units()
108
109     report.info('\n')
110     for unit in units:
111         if unit.state == rp.FAILED:
112             report.plain('  * %s: %s, exit: %3s, err: %s' \
113                    % (unit.uid, unit.state[:4],
114                       unit.exit_code, unit.stderr.strip()[-35:]))
115             report.error('>>err\n')
116         else:
117             report.plain('  * %s: %s, exit: %3s, out: %s' \
118                    % (unit.uid, unit.state[:4],
119                       unit.exit_code, unit.stdout.strip()[:35]))
120             report.ok('>>ok\n')
121
```

Instead of running an executable we are almost certain will succeed, we now and then insert an intentional faulty one whose specified executable file does not exist on the target system. Upon state inspection, we expect to find a *FAILED* state for those units, and a respective informative *stderr* output:

Running the Example

Running the example will result in an output similar to the one shown below:



Note: You will see red glyphs during the result gathering phase, indicating that a failed unit has been collected. The example output above also demonstrates an important feature: execution ordering of units is *not preserved*, that order

is independent of the order of submission. Any unit dependencies need to be resolved on application level!

What's Next?

The next user guide section (*Use Multiple Pilots* (page 20)) will return to the basic example (ie. no failing units are expected), but will now submit those units to more than one concurrent pilots.

1.4.4 Use Multiple Pilots

We have seen in the previous examples how an RP pilot acts as a container for multiple compute unit executions. There is in principle no limit on how many of those pilots are used to execute a specific workload, and specifically, pilot don't need to run on the same resource!

`03_multiple_pilots.py` demonstrates that, and features the following diff to the previous examples:

```
55      # Define an [n]-core local pilot that runs for [x] minutes
56      # Here we use a dict to initialize the description object
57      report.info('create pilot description')
58      pd_init = {
59              'resource'      : resource,
60              'cores'         : 64,  # pilot size
61              'runtime'       : 10,  # pilot runtime (min)
62              'project'       : config[resource]['project'],
63              'queue'         : config[resource]['queue'],
64              'access_schema' : config[resource]['schema']
65              }
66      pdesc = rp.ComputePilotDescription(pd_init)
----------------------------------------------------------------------------
67      report.ok('>>ok\n')
68
69      # Launch the pilot.
70      pilot = pmgr.submit_pilots(pdesc)
71
72
73      report.header('submit units')
74
75      # Register the ComputePilot in a UnitManager object.
76      umgr = rp.UnitManager(session=session)
77      umgr.add_pilots(pilot)
78
```

```
54      # Define an [n]-core local pilot that runs for [x] minutes
55      # Here we use a dict to initialize the description object
56      pdescs = list()
57      report.info('create pilot descriptions')
58      for resource in resources:
59          pd_init = {
60                  'resource'      : resource,
61                  'cores'         : 64,  # pilot size
62                  'runtime'       : 10,  # pilot runtime (min)
63                  'project'       : config[resource]['project'],
64                  'queue'         : config[resource]['queue'],
65                  'access_schema' : config[resource]['schema']
66                  }
67          pdescs.append(rp.ComputePilotDescription(pd_init))
68      report.ok('>>ok\n')
69
70      # Launch the pilots.
71      pilots = pmgr.submit_pilots(pdescs)
72
73
74      report.header('submit units')
75
76      # Register the ComputePilot in a UnitManager object.
77      umgr = rp.UnitManager(session=session)
78      umgr.add_pilots(pilots)
79
```

Instead of creating *one* pilot description, we here create one for any resource specified as command line parameter, no matter if those parameters point to the same resource targets or not.

The units are distributed over the created set of pilots according to some scheduling mechanism – section *Selecting a Unit Scheduler* (page 22) will discuss how an application can choose between different scheduling policies. The default policy used here is *Round Robin*.

Running the Example

The workload of our example has now changed to report the respectively used pilot on stdout, and the output shows that. We here exemplarily start a pilot on *local.localhost*, and one on *xsede.stampede*:

```
$ ./examples/getting_started_03.py local.localhost xsede.stampede

================================================================================
 Getting Started (RP version v0.36)
================================================================================

create session rp.session.cameo.merzky.016721.0001                          ok
read config                                                                 ok

--------------------------------------------------------------------------------
submit pilots

create pilot manager                                                        ok
create pilot descriptions                                                    \
create pilot description [local.localhost:64]                               ok
create pilot description [xsede.stampede:64]                                ok
                                                                            ok
submit 2 pilot(s) ..                                                        ok

--------------------------------------------------------------------------------
submit units

create unit manager                                                         ok
add 2 pilot(s)                                                              ok
create 128 unit description(s)
       ..................................................................
       ..........................................................           ok
submit 128 unit(s)
       ..................................................................
       ..........................................................           ok

--------------------------------------------------------------------------------
gather results

wait for 128 unit(s)
       ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++|
       ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++           ok

  * unit.000000: Done, exit:   0, out: pilot.0000
  * unit.000001: Done, exit:   0, out: pilot.0001
  * unit.000002: Done, exit:   0, out: pilot.0000
  * unit.000003: Done, exit:   0, out: pilot.0001
  * [...]
  * unit.000125: Done, exit:   0, out: pilot.0001
  * unit.000126: Done, exit:   0, out: pilot.0000
  * unit.000127: Done, exit:   0, out: pilot.0001

--------------------------------------------------------------------------------
finalize

closing session rp.session.cameo.merzky.016721.0001                          \
close pilot manager                                                          \
wait for 2 pilot(s) **                                                      ok
                                                                            ok
close unit manager                                                          ok
                                                                            ok

--------------------------------------------------------------------------------
```

What's Next?

Using multiple pilots is very powerful – it becomes more powerful if you allow RP to load-balance units between them. *Selecting a Unit Scheduler* (page 22) will show how to do just that.

1.4.5 Selecting a Unit Scheduler

We have seen in the previous examples how the `radical.pilot.UnitManager` (page 49) matches submitted units to pilots for execution. On constructing the unit manager, it can be configured to use a specific scheduling policy for that. The following policies are implemented:

- *rp.SCHEDULER_ROUND_ROBIN*: alternate units between all available pilot. This policy leads to a static and fair, but not necessarily load-balanced unit assignment.
- *rp.SCHEDULER_BACKFILLING*: dynamic unit scheduling based on pilot capacity and availability. This is the most intelligent scheduler with good load balancing, but it comes with a certain scheduling overhead.

An important element to consider when discussing unit scheduling is pilot startup time: pilot jobs can potentially sit in batch queues for a long time, or pass quickly, depending on their size and resource usage, etc. Any static assignment of units will not be able to take that into account – and the first pilot may have finished all its work before a second pilot even came up.

This is what the backfilling scheduler tries to address: it only schedules units once the pilot is available, and only as many as a pilot can execute at any point in time. As this requires close communication between pilot and scheduler, that scheduler will incur a runtime overhead for each unit – so that is only advisable for heterogeneous workloads and/or pilot setups, and for long running units.

`04_scheduler_selection.py` shows an exemplary scheduling selector, with the following diff to the previous multi-pilot example:

```
$ ./examples/getting_started_04.py local.localhost xsede.stampede epsrc.archer

================================================================================
 Getting Started (RP version v0.36)
================================================================================

create session rp.session.cameo.merzky.016721.0004                          ok
read config                                                                 ok

--------------------------------------------------------------------------------
submit pilots

create pilot manager                                                        ok
create pilot descriptions                                                    \
create pilot description [local.localhost:64]                               ok
create pilot description [xsede.stampede:64]                                ok
create pilot description [epsrc.archer:64]                                  ok
                                                                            ok
submit 3 pilot(s) ...                                                       ok

--------------------------------------------------------------------------------
submit units

select scheduler                                                backfilling
create unit manager                                                         ok
add 3 pilot(s)                                                              ok
create 128 unit description(s)
       ..................................................................
       ..........................................................           ok
submit 128 unit(s)
       ..................................................................
       ..........................................................           ok

--------------------------------------------------------------------------------
gather results

wait for 128 unit(s)
       ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++|
       ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++           ok

  * unit.000000: Done, exit:   0, out: pilot.0000
  * unit.000001: Done, exit:   0, out: pilot.0000
  * unit.000002: Done, exit:   0, out: pilot.0000
  * [...]
  * unit.000125: Done, exit:   0, out: pilot.0000
  * unit.000126: Done, exit:   0, out: pilot.0000
  * unit.000127: Done, exit:   0, out: pilot.0000

--------------------------------------------------------------------------------
finalize

closing session rp.session.cameo.merzky.016721.0004                          \
close pilot manager                                                          \
wait for 3 pilot(s) +-*                                                     ok
                                                                            ok
close unit manager                                                          ok
                                                                            ok

--------------------------------------------------------------------------------
```

It will select *Round Robin* scheduling for two pilots, and *Backfilling* for three or more.

Running the Example

We show here the output for 3 pilots, where one is running locally (and thus is likely to come up quickly), and 2 are running exemplarily on *xsede.stampede* and *epsrc.archer*, respectively, where they likely will sit in the queue for a bit. We thus expect the backfilling scheduler to prefer the local pilot (*pilot.0000*).

What's Next?

Using multiple pilots is very powerful – it becomes more powerful if you allow RP to load-balance units between them. *Selecting a Unit Scheduler* (page 22) will show how to do just that.

1.4.6 Staging Unit Input Data

The vast majority of applications operate on data, and many of those read input data from files. Since RP provides an abstraction above the resource layer, it can run a ComputeUnit on any pilot the application created (see *Selecting a Unit Scheduler* (page 22)). To ensure that the CU finds the data it needs on the resource where it runs, RP provides a mechanism to stage input data automatically.

For each compute unit, the application can specify

- *source*: what data files need to be staged;
- *target*: what should the path be in the context of the CU execution;
- *action*: how should data be staged.

If *source* and *target* file names are the same, and if *action* is the default *rp.TRANSFER*, then you can simply specify CU input data by giving a list of file names (we'll discuss more complex staging directives in a *later* (page 26) example):

```
cud = rp.ComputeUnitDescription()
cud.executable     = '/usr/bin/wc'
cud.arguments      = ['-c', 'input.dat']
cud.input_staging  = ['input.dat']
```

`05_unit_input_data.py` contains an example application which uses the above code block. It otherwise does not differ from our earlier examples (but only adds on-th-fly creation of *input.dat*).

Running the Example

The result of this example's execution is straight forward, as expected, but proves that the file staging happened as planned. You will likely notice though that the code runs significantly longer than earlier ones, because of the file staging overhead – we will discuss in *Sharing Unit Input Data* (page 28) how file staging can be optimized for units which share the same input data.

```
================================================================================
 Getting Started (RP version v0.37)
================================================================================

create session rp.session.cameo.merzky.016725.0035                          ok
read config                                                                 ok

--------------------------------------------------------------------------------
submit pilots

create pilot manager                                                        ok
create pilot description                                                     \
create pilot description [local.localhost:64]                               ok
                                                                            ok
submit 1 pilot(s) .                                                         ok

--------------------------------------------------------------------------------
submit units

create unit manager                                                         ok
add 1 pilot(s)                                                              ok
create 128 unit description(s)
       ..................................................................
       ..........................................................           ok
submit 128 unit(s)
       ..................................................................
       ..........................................................           ok

--------------------------------------------------------------------------------
gather results

wait for 128 unit(s)
       ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++|
       ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++           ok

  * unit.000000: Done, exit:   0, out: 36 input.dat
  * unit.000001: Done, exit:   0, out: 36 input.dat
  * unit.000002: Done, exit:   0, out: 36 input.dat
  * [...]
  * unit.000125: Done, exit:   0, out: 36 input.dat
  * unit.000126: Done, exit:   0, out: 36 input.dat
  * unit.000127: Done, exit:   0, out: 36 input.dat

--------------------------------------------------------------------------------
finalize

closing session rp.session.cameo.merzky.016725.0035                          \
close pilot manager                                                          \
wait for 1 pilot(s) *                                                       ok
                                                                            ok
close unit manager                                                          ok
                                                                            ok


--------------------------------------------------------------------------------
```

What's Next?

The obvious next step will be to handle output data: *Staging Unit Output Data* (page 26) will address exactly this, and also provide some more details on different modes of data staging, before *Sharing Unit Input Data* (page 28) will introduce RP's capability to share data between different compute units.

1.4.7 Staging Unit Output Data

Upon completion, CUs have often creates some amount of data. We have seen in *Obtaining Unit Details* (page 16) how we can inspect the CU's `stdout` string – but that will not be useful beyond the most trivial workloads. This section introduces how created data can be staged back to the RP application, and/or staged to 3rd party storage.

Output staging is in principle specified just like the input staging discussed in the ref:*previous <chapter_user_guide_05>* section:

- `source`: what data files need to be staged from the context of the finished CU;
- `target`: where should the data be staged to;
- `action`: how should data be staged.

In this example we actually use the long form, and specify the output file name to be changed to a unique name during staging:

```
for i in range(0, n):
    cud.executable     = '/bin/cp'
    cud.arguments      = ['-v', 'input.dat', 'output.dat']
    cud.input_staging  = ['input.dat']
    cud.output_staging = {'source': 'output.dat',
                          'target': 'output_%03d.dat' % i,
                          'action': rp.TRANSFER}
```

`06_unit_output_data.py` contains an example application which uses the above code block. It otherwise does not significantly differ from our previous example.

Running the Example

The result of this example's execution shows that the output files have been renamed during the output-staging phase:

```
================================================================================
 Getting Started (RP version v0.37)
================================================================================

create session rp.session.cameo.merzky.016725.0044                          ok
read config                                                                 ok

--------------------------------------------------------------------------------
submit pilots

create pilot manager                                                        ok
create pilot description                                                     \
create pilot description [local.localhost:64]                               ok
                                                                            ok
submit 1 pilot(s) .                                                         ok

--------------------------------------------------------------------------------
submit units

create unit manager                                                         ok
add 1 pilot(s)                                                              ok
create 128 unit description(s)
       ..................................................................
       ..........................................................           ok
submit 128 unit(s)
       ..................................................................
       ..........................................................           ok

--------------------------------------------------------------------------------
gather results

wait for 128 unit(s)
       ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++|
       ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++           ok

  * unit.000000: Done, exit:   0, out: 'input.dat' -> 'output.dat'
  * unit.000001: Done, exit:   0, out: 'input.dat' -> 'output.dat'
  * unit.000002: Done, exit:   0, out: 'input.dat' -> 'output.dat'
  * [...]
  * unit.000125: Done, exit:   0, out: 'input.dat' -> 'output.dat'
  * unit.000126: Done, exit:   0, out: 'input.dat' -> 'output.dat'
  * unit.000127: Done, exit:   0, out: 'input.dat' -> 'output.dat'

resulting data files:

output_000.dat  output_026.dat  output_052.dat  output_078.dat  output_104.dat
output_001.dat  output_027.dat  output_053.dat  output_079.dat  output_105.dat
output_002.dat  output_028.dat  output_054.dat  output_080.dat  output_106.dat
output_003.dat  output_029.dat  output_055.dat  output_081.dat  output_107.dat
[...]
output_023.dat  output_049.dat  output_075.dat  output_101.dat  output_127.dat
output_024.dat  output_050.dat  output_076.dat  output_102.dat
output_025.dat  output_051.dat  output_077.dat  output_103.dat

--------------------------------------------------------------------------------
finalize

closing session rp.session.cameo.merzky.016725.0044                          \
close pilot manager                                                          \
wait for 1 pilot(s) *                                                       ok
                                                                            ok
close unit manager                                                          ok
                                                                            ok

--------------------------------------------------------------------------------
```

What's Next?

As we are now comfortable with input and output staging, we will next look into an optimization which is important for a large set of use cases: the *sharing of input data* (page 28) between multiple compute units.

1.4.8 Sharing Unit Input Data

RP aims to support the concurrent execution of many tasks, and for many workloads which fit that broad description, those tasks share (some or all) input data. We have seen *earlier* (page 24) that input staging can incur a significant runtime overhead – but that can be significantly reduced by avoiding redundant staging operations.

For this purpose, each RP *pilot* manages a spaces of shared data, and any data put into that space by the application can later be symlinked into the unit's workdir, for consumption:

```
# stage shared data from `pwd` to the pilot's shared staging space
pilot.stage_in({'source': 'file://%s/input.dat' % os.getcwd(),
                'target': 'staging:///input.dat',
                'action': rp.TRANSFER})

[...]

for i in range(0, n):

    cud = rp.ComputeUnitDescription()

    cud.executable     = '/usr/bin/wc'
    cud.arguments      = ['-c', 'input.dat']
    cud.input_staging  = {'source': 'staging:///input.dat',
                          'target': 'input.dat',
                          'action': rp.LINK
                         }
```

The *rp.LINK* staging action requests a symlink to be created by RP, instead of the copy operation used on the default *rp.TRANSFER* action. The full example can be found here: `07_shared_unit_data.py`.

Note: Unlike many other methods in RP, the *pilot.stage_in* option is *synchronous*, ie. it will only return once the transfer has been completed. That semantics may change in a future version of RP.

Running the Example

The result of this example's execution is the very same as we saw in the *previous* (page 24), but it will now run significantly faster due to the removed staging redundancy (at least for non-local pilots):

```
================================================================================
 Getting Started (RP version v0.37)
================================================================================

create session rp.session.cameo.merzky.016725.0056                          ok
read config                                                                 ok

--------------------------------------------------------------------------------
submit pilots

create pilot manager                                                        ok
create pilot description                                                     \
create pilot description [local.localhost:64]                               ok
                                                                            ok
submit 1 pilot(s) .                                                         ok
stage shared data                                                           ok

--------------------------------------------------------------------------------
submit units

create unit manager                                                         ok
add 1 pilot(s)                                                              ok
create 128 unit description(s)
       ................................................................................
       ..........................................................           ok
submit 128 unit(s)
       ................................................................................
       ..........................................................           ok

--------------------------------------------------------------------------------
gather results

wait for 128 unit(s)
       ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++|
       ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++ ok

  * unit.000000: Done, exit:   0, out: 36 input.dat
  * unit.000001: Done, exit:   0, out: 36 input.dat
  * unit.000002: Done, exit:   0, out: 36 input.dat
  * [...]
  * unit.000125: Done, exit:   0, out: 36 input.dat
  * unit.000126: Done, exit:   0, out: 36 input.dat
  * unit.000127: Done, exit:   0, out: 36 input.dat

--------------------------------------------------------------------------------
finalize

closing session rp.session.cameo.merzky.016725.0056                          \
close pilot manager                                                          \
wait for 1 pilot(s) *                                                       ok
                                                                            ok
close unit manager                                                          ok
session lifetime: 58.2s                                                     ok

--------------------------------------------------------------------------------
```

What's Next?

This completes the discussion on data staging – the next sections will go into more details of the units execution: *environment setup* (page 30), *pre- and post- execution* (page 34), and *MPI applications* (page 32).

1.4.9 Setup Unit Environment

Different Applications come with different requirements to the runtime environment. This section will describe how the shell environment for a CU can be configured, the next two sections will describe how to *configure CUs to run as MPI application* (page 32) and how to *insert arbitrary setup commands* (page 34).

The CU environment is simply configured as a Python dictionary on the unit description, like this:

```
cud = rp.ComputeUnitDescription()

cud.executable  = '/bin/echo'
cud.arguments   = ['$RP_UNIT_ID greets $TEST']
cud.environment = {'TEST' : 'jabberwocky'}
```

which will make the environment variable *TEST* available during CU execution. Some other variables, such as the *RP_UNIT_ID* above, are set by RP internally and are here used for demonstration – but those should not be relied upon.

Running the Example

`08_unit_environment.py`. uses the above blob to run a bag of *echo* commands:

```
================================================================================
 Getting Started (RP version v0.37)
================================================================================

create session rp.session.cameo.merzky.016725.0065                          ok
read config                                                                 ok

--------------------------------------------------------------------------------
submit pilots

create pilot manager                                                        ok
create pilot description                                                     \
create pilot description [local.localhost:64]                               ok
                                                                            ok
submit 1 pilot(s) .                                                         ok

--------------------------------------------------------------------------------
submit units

create unit manager                                                         ok
add 1 pilot(s)                                                              ok
create 128 unit description(s)
       ..................................................................
       ..........................................................           ok
submit 128 unit(s)
       ..................................................................
       ..........................................................           ok

--------------------------------------------------------------------------------
gather results

wait for 128 unit(s)
       +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++|
       ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++           ok

  * unit.000000: Done, exit:   0, out: unit.000000 greets jabberwocky
  * unit.000001: Done, exit:   0, out: unit.000001 greets jabberwocky
  * unit.000002: Done, exit:   0, out: unit.000002 greets jabberwocky
  * [...]
  * unit.000125: Done, exit:   0, out: unit.000125 greets jabberwocky
  * unit.000126: Done, exit:   0, out: unit.000126 greets jabberwocky
  * unit.000127: Done, exit:   0, out: unit.000127 greets jabberwocky

--------------------------------------------------------------------------------
finalize

closing session rp.session.cameo.merzky.016725.0065                          \
close pilot manager                                                          \
wait for 1 pilot(s) *                                                       ok
                                                                            ok
close unit manager                                                          ok
session lifetime: 118.2s                                                    ok

--------------------------------------------------------------------------------
```

What's Next?

Running MPI applications (page 32), and :ref:providing more generic environment setup <chapter_user_guide_10>‘,

are the topics for the next two sections.

1.4.10 MPI Applications

CUs which execute MPI applications are, from an RP application perspective, not really different from other CUs – but the application needs to communicate to RP that the unit will (a) allocate a number of cores, and (b) needs to be started under an MPI regime. The respective CU description entries are shown below:

```
cud = rp.ComputeUnitDescription()

cud.executable  = '/bin/echo'
cud.arguments   = ['-n', '$RP_UNIT_ID ']
cud.cores       = 2
cud.mpi         = True
```

This example should result in the unit ID echo'ed *twice*, once per MPI rank.

Note: Some RP configurations require MPI applications to be linked against a specific version of OpenMPI. This is the case when using *orte* or *orte_lib* launch methods in the agent. Please contact the mailing list if you need support with relinking your application.

Running the Example

`09_mpi_units.py`. uses the above blob to run a bag of duplicated *echo* commands:

```
================================================================================
 Getting Started (RP version v0.37)
================================================================================

create session rp.session.cameo.merzky.016725.0066                          ok
read config                                                                 ok

--------------------------------------------------------------------------------
submit pilots

create pilot manager                                                        ok
create pilot description                                                     \
create pilot description [local.localhost:64]                               ok
                                                                            ok
submit 1 pilot(s) .                                                         ok

--------------------------------------------------------------------------------
submit units

create unit manager                                                         ok
add 1 pilot(s)                                                              ok
create 128 unit description(s)
       ..................................................................
       ..........................................................           ok
submit 128 unit(s)
       ..................................................................
       ..........................................................           ok

--------------------------------------------------------------------------------
gather results

wait for 128 unit(s)
       ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++|
       ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++           ok

  * unit.000000: Done, exit:   0, out: unit.000000 unit.000000
  * unit.000001: Done, exit:   0, out: unit.000001 unit.000001
  * unit.000002: Done, exit:   0, out: unit.000002 unit.000002
  * [...]
  * unit.000125: Done, exit:   0, out: unit.000125 unit.000125
  * unit.000126: Done, exit:   0, out: unit.000126 unit.000126
  * unit.000127: Done, exit:   0, out: unit.000127 unit.000127

--------------------------------------------------------------------------------
finalize

closing session rp.session.cameo.merzky.016725.0066                          \
close pilot manager                                                          \
wait for 1 pilot(s) *                                                       ok
                                                                            ok
close unit manager                                                          ok
session lifetime: 93.8s                                                     ok

--------------------------------------------------------------------------------
```

What's Next?

Running MPI applications (page 32), and :ref:providing more generic environment setup <chapter_user_guide_10>‘,

are the topics for the next two sections.

1.4.11 Using Pre- and Post- exec commands

In some cases, applications (and thus CUs) need more complex and customizable setup routines than can be expressed via *environment* (page 30) or *MPI flags* (page 32). A frequent example is the use of *module load* commands on various HPC resources, which are used to prepare application runtime environments in a well defined, system specific way.

RP supports the invocation of such commands via the *pre_exec* and *post_exec* keys for the CU descriptions.

Note: Pre- and Post- execution is performed on the *resource headnode* – abuse of these commands for any compute or I/O heavy loads can lead to serious consequences, and will likely draw the wrath of the system administrators upon you! You have been warned...

The code example below exemplarily demonstrates the same environment setup we have been using in an earlier section, but now rendered via an *pre_exec* command:

```
cud = rp.ComputeUnitDescription()

cud.pre_exec    = ['export TEST=jabberwocky']
cud.executable  = '/bin/echo'
cud.arguments   = ['$RP_UNIT_ID greets $TEST']
```

which again will make the environment variable *TEST* available during CU execution.

Running the Example

`10_pre_and_post_exec.py`. uses the above blob to run a bag of *echo* commands:

```
================================================================================
 Getting Started (RP version v0.37)
================================================================================

create session rp.session.cameo.merzky.016725.0067                          ok
read config                                                                 ok

--------------------------------------------------------------------------------
submit pilots

create pilot manager                                                        ok
create pilot description                                                     \
create pilot description [local.localhost:64]                               ok
                                                                            ok
submit 1 pilot(s) .                                                         ok

--------------------------------------------------------------------------------
submit units

create unit manager                                                         ok
add 1 pilot(s)                                                              ok
create 128 unit description(s)
       ..................................................................
       ..........................................................           ok
submit 128 unit(s)
       ..................................................................
       ..........................................................           ok

--------------------------------------------------------------------------------
gather results

wait for 128 unit(s)
       ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++|
       ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++           ok

  * unit.000000: Done, exit:   0, out: unit.000000 greets jabberwocky
  * unit.000001: Done, exit:   0, out: unit.000001 greets jabberwocky
  * unit.000002: Done, exit:   0, out: unit.000002 greets jabberwocky
  * [...]
  * unit.000125: Done, exit:   0, out: unit.000125 greets jabberwocky
  * unit.000126: Done, exit:   0, out: unit.000126 greets jabberwocky
  * unit.000127: Done, exit:   0, out: unit.000127 greets jabberwocky

--------------------------------------------------------------------------------
finalize

closing session rp.session.cameo.merzky.016725.0067                          \
close pilot manager                                                          \
wait for 1 pilot(s) *                                                       ok
                                                                            ok
close unit manager                                                          ok
session lifetime: 122.9s                                                    ok

--------------------------------------------------------------------------------
```

What's Next?

The RP User-Guide concludes with this section. We recommend to check out the RP API documentation next, and use it to write an RP application to run your own workload. Ii is easiest to do so by starting off with the `canonical example`, and then add bits and pieces from the various user :ref:<chapter_user_guide>‘ sections as needed.

1.5 Examples

This chapter contains a set of more elaborate examples which demonstrate various features of RADICAL-Pilot in more realistic environments.

Contents:

1.5.1 Simple Bag-of-Tasks

You might be wondering how to create your own RADICAL-Pilot script or how RADICAL-Pilot can be useful for your needs. Before delving into the remote job and data submission capabilities that RADICAL-Pilot has, its important to understand the basics.

The simplest usage of a pilot-job system is to submit multiple identical tasks (a ‘Bag of Tasks’ / ‘BoT’) collectively, i.e. as one big job! Such usage arises for example to perform parameter sweep jobs, or to execute ensemble simulation.

We will create an example which submits *N* tasks using RADICAL-Pilot. The tasks are all identical, except that they each record their number in their output. This type of run is very useful if you are running many jobs using the same executable (but perhaps with different input files). Rather than submit each job individually to the queuing system and then wait for every job to become active individually, you submit just one container job (called the Pilot). When this pilot becomes active, it executes your tasks on the reserved cores. RADICAL-Pilot names such tasks ‘Compute Units’, or short ‘CUs’.

Preparation

Download the file `simple_bot.py` with the following command:

```
curl -O https://raw.githubusercontent.com/radical-cybertools/radical.pilot/master/
↪examples/docs/simple_bot.py
```

Open the file `simple_bot.py` with your favorite editor. The example should work right out of the box on your local machine. However, if you want to try it out with different resources, like remote HPC clusters, look for the sections marked:

```
# ----- CHANGE THIS -- CHANGE THIS -- CHANGE THIS -- CHANGE THIS ------
```

and change the code below accordging to the instructions in the comments.

This assumes you have installed RADICAL-Pilot either globally or in a Python virtualenv. You also need access to a MongoDB server.

Set the *RADICAL_PILOT_DBURL* environment variable in your shell to the MongoDB server you want to use, for example:

```
export RADICAL_PILOT_DBURL=mongodb://<user>:<pass>@<mongodb_server>:27017/<database>
```

If RADICAL-Pilot is installed and the MongoDB URL is set, you should be good to run your program (the database is created on the fly):

```
python simple_bot.py
```

The output should look something like this:

```
Initializing Pilot Manager ...
Submitting Compute Pilot to Pilot Manager ...
Initializing Unit Manager ...
Registering Compute Pilot with Unit Manager ...
Submit Compute Units to Unit Manager ...
Waiting for CUs to complete ...
...
Waiting for CUs to complete ...
All CUs completed!
Closed session, exiting now ...
```

Logging and Debugging

Since working with distributed systems is inherently complex and much of the complexity is hidden within RADICAL-Pilot, it is necessary to do a lot of internal logging. By default, logging output is disabled, but if something goes wrong or if you're just curious, you can enable the logging output by setting the environment variable `RADICAL_PILOT_VERBOSE` to a value between CRITICAL (print only critical messages) and DEBUG (print all messages). For more details on logging, see under 'Debugging' in chapter *Developer Documentation* (page 74).

Give it a try with the above example:

```
RADICAL_PILOT_VERBOSE=DEBUG python simple_bot.py
```

1.5.2 Chained Tasks

What if you had two different executables – A and B, to run? What if this second set of executables (B) had some dependencies on data from the first set (A)? Can you use one RADICAL-Pilot to run both set jobs? Yes!

The example below submits a set of echo jobs (set A) using RADICAL-Pilot, and for every successful job (with state `DONE`), it submits another job (set B) to the same Pilot-Job.

We can think of A is being comprised of subjobs {a1,a2,a3}, while B is comprised of subjobs {b1,b2,b3}. Rather than wait for each subjob {a1},{a2},{a3} to complete, {b1} can run as soon as {a1} is complete, or {b1} can run as soon as a slot becomes available – i.e. {a2} could finish before {a1}.

The code below demonstrates this behavior. As soon as there is a slot available to run a job in B (i.e. a job in A has completed), it executes the job in B. This keeps the RADICAL-Pilot throughput high.

Preparation

Download the file `chained_tasks.py` with the following command:

```
curl -O https://raw.githubusercontent.com/radical-cybertools/radical.pilot/master/
↪examples/docs/chained_tasks.py
```

Open the file `chained_tasks.py` with your favorite editor. The example should work right out of the box on your local machine. However, if you want to try it out with different resources, like remote HPC clusters, look for the sections marked:

```
# ----- CHANGE THIS -- CHANGE THIS -- CHANGE THIS -- CHANGE THIS ------
```

and change the code below accordging to the instructions in the comments.

Execution

This assumes you have installed RADICAL-Pilot either globally or in a Python virtualenv. You also need access to a MongoDB server.

Set the *RADICAL_PILOT_DBURL* environment variable in your shell to the MongoDB server you want to use, for example:

```
export RADICAL_PILOT_DBURL=mongodb://<user>:<pass>@<mongodb_server>:27017/
```

If RADICAL-Pilot is installed and the MongoDB URL is set, you should be good to run your program:

```
python chained_tasks.py
```

1.5.3 Coupled Tasks

The script is a simple workflow which submits a set of tasks A and set of tasks B and waits until they are completed before submiting a set of tasks C. It demonstrates synchronization mechanisms provided by the Pilot-API. This example is useful if a task in C has dependencies on some of the output generated from tasks in A and B.

Preparation

Download the file `coupled_tasks.py` with the following command:

```
curl -O https://raw.githubusercontent.com/radical-cybertools/radical.pilot/master/
↪examples/docs/coupled_tasks.py
```

Open the file `coupled_tasks.py` with your favorite editor. The example should work right out of the box on your local machine. However, if you want to try it out with different resources, like remote HPC clusters, look for the sections marked:

```
# ----- CHANGE THIS -- CHANGE THIS -- CHANGE THIS -- CHANGE THIS ------
```

and change the code below accordging to the instructions in the comments.

You will need to make the necessary changes to `coupled_tasks.py` as you did in the previous example. The important difference between this file and the previous file is that there are three separate "USER DEFINED CU DESCRIPTION" sections - numbered 1-3. Again, these two sections will not require any modifications for the purposes of this tutorial. We will not review every variable again, but instead, review the relationship between the 3 task descriptions. The three task descriptions are identical except that they each have a different CU_SET variable assigned - either A, B, or C.

NOTE that we call each task set the same number of times (i.e. NUMBER_JOBS) in the tutorial code, but this is not a requirement. It just simplifies the code for tutorial purposes. It is possible you want to run 16 A, 16 B, and then 32 C using the output from both A and B.

In this case, the important logic to draw your attention too is around line 140:

```
print "Waiting for 'A' and 'B' CUs to complete..."
umgr.wait_units()
print "Executing 'C' tasks now..."
```

In this example, we submit both the A and B tasks to the Pilot, but instead of running C tasks right away, we call `wait()` on the unit manager. This tells RADICAL-Pilot to wait for all of the submitted tasks to finish, before continuing in the code. After all the A and B (submitted tasks) have finished, it then submits the C tasks.

Execution

This assumes you have installed RADICAL-Pilot either globally or in a Python virtualenv. You also need access to a MongoDB server.

Set the *RADICAL_PILOT_DBURL* environment variable in your shell to the MongoDB server you want to use, for example:

```
export RADICAL_PILOT_DBURL=mongodb://<user>:<pass>@<mongodb_server>:27017/
```

If RADICAL-Pilot is installed and the MongoDB URL is set, you should be good to run your program:

```
python coupled_tasks.py
```

1.5.4 MPI tasks

So far we have run a sequential tasks in a number of configurations.

This example introduces two new concepts: running multi-core MPI tasks and specifying input data for the task, in this case a simple python MPI script.

Preparation

Download the file `mpi_tasks.py` with the following command:

```
curl -O https://raw.githubusercontent.com/radical-cybertools/radical.pilot/master/
↪examples/docs/mpi_tasks.py
```

Open the file `mpi_tasks.py` with your favorite editor. The example might work right out of the box on your local machine, this depends whether you have a local MPI installation. However, if you want to try it out with different resources, like remote HPC clusters, look for the sections marked:

```
# ----- CHANGE THIS -- CHANGE THIS -- CHANGE THIS -- CHANGE THIS ------
```

and change the code below according to the instructions in the comments.

This example makes use of an application that we first download to our own environment and then have staged as input to the MPI tasks.

Download the file `helloworld_mpi.py` with the following command:

```
curl -O https://raw.githubusercontent.com/radical-cybertools/radical.pilot/master/
↪examples/helloworld_mpi.py
```

Execution

** This assumes you have installed RADICAL-Pilot either globally or in a Python virtualenv. You also need access to a MongoDB server.**

Set the *RADICAL_PILOT_DBURL* environment variable in your shell to the MongoDB server you want to use, for example:

```
export RADICAL_PILOT_DBURL=mongodb://<user>:<pass>@<mongodb_server>:27017/
```

If RADICAL-Pilot is installed and the MongoDB URL is set, you should be good to run your program:

```
python mpi_tasks.py
```

The output should look something like this:

```
Initializing Pilot Manager ...
Submitting Compute Pilot to Pilot Manager ...
Initializing Unit Manager ...
Registering Compute Pilot with Unit Manager ...
Submit Compute Units to Unit Manager ...
Waiting for CUs to complete ...
...
Waiting for CUs to complete ...
All CUs completed successfully!
Closed session, exiting now ...
```

Logging and Debugging

Since working with distributed systems is inherently complex and much of the complexity is hidden within RADICAL-Pilot, it is necessary to do a lot of internal logging. By default, logging output is disabled, but if something goes wrong or if you're just curious, you can enable the logging output by setting the environment variable `RADICAL_PILOT_VERBOSE` to a value between CRITICAL (print only critical messages) and DEBUG (print all messages).

Give it a try with the above example:

```
RADICAL_PILOT_VERBOSE=DEBUG python simple_bot.py
```

1.6 API Reference

1.6.1 Sessions and Security Contexts

Sessions

class radical.pilot.**Session**(*dburl=None*, *uid=None*, *cfg=None*, *_connect=True*)

A Session encapsulates a RADICAL-Pilot instance and is the *root* object

A Session holds *radical.pilot.PilotManager* (page 43) and *radical.pilot.UnitManager* (page 49) instances which in turn hold *radical.pilot.ComputePilot* (page 46) and *radical.pilot.ComputeUnit* (page 55) instances.

__init__(*dburl=None*, *uid=None*, *cfg=None*, *_connect=True*)

Creates a new session. A new Session instance is created and stored in the database.

Arguments:

- **dburl** (*string*): The MongoDB URL. If none is given, RP uses the environment variable RADICAL_PILOT_DBURL. If that is not set, an error will be raises.
- **uid** (*string*): Create a session with this UID. *Only use this when you know what you are doing!*

Returns:

- A new Session instance.

Raises:

- *radical.pilot.DatabaseError* (page 57)

close(*cleanup=False*, *terminate=True*, *download=False*)
Closes the session.

All subsequent attempts access objects attached to the session will result in an error. If cleanup is set to True (default) the session data is removed from the database.

Arguments:

- **cleanup** (*bool*): Remove session from MongoDB (implies * terminate)
- **terminate** (*bool*): Shut down all pilots associated with the session.

Raises:

- `radical.pilot.IncorrectState` if the session is closed or doesn't exist.

as_dict()
Returns a Python dictionary representation of the object.

created
Returns the UTC date and time the session was created.

connected
Returns the most recent UTC date and time the session was reconnected to.

closed
Returns the time of closing

inject_metadata(*metadata*)
Insert (experiment) metadata into an active session RP stack version info always get added.

list_pilot_managers()
Lists the unique identifiers of all *radical.pilot.PilotManager* (page 43) instances associated with this session.

Returns:

- A list of *radical.pilot.PilotManager* (page 43) uids (*list* of *strings*).

get_pilot_managers(*pmgr_uids=None*)
returns known PilotManager(s).

Arguments:

- **pmgr_uids** [*string*]: unique identifier of the PilotManager we want

Returns:

- One or more [*radical.pilot.PilotManager* (page 43)] objects.

list_unit_managers()
Lists the unique identifiers of all *radical.pilot.UnitManager* (page 49) instances associated with this session.

Returns:

- A list of *radical.pilot.UnitManager* (page 49) uids (*list* of *strings*).

get_unit_managers(*umgr_uids=None*)
returns known UnitManager(s).

Arguments:

- **umgr_uids** [*string*]: unique identifier of the UnitManager we want

Returns:

- One or more [*radical.pilot.UnitManager* (page 49)] objects.

list_resources()
Returns a list of known resource labels which can be used in a pilot description. Not that resource aliases won't be listed.

add_resource_config(*resource_config*)
Adds a new `radical.pilot.ResourceConfig` to the PilotManager's dictionary of known resources, or accept a string which points to a configuration file.

For example:

```
rc = radical.pilot.ResourceConfig(label="mycluster")
rc.job_manager_endpoint = "ssh+pbs://mycluster
rc.filesystem_endpoint  = "sftp://mycluster
rc.default_queue        = "private"
rc.bootstrapper         = "default_bootstrapper.sh"

pm = radical.pilot.PilotManager(session=s)
pm.add_resource_config(rc)

pd = radical.pilot.ComputePilotDescription()
pd.resource = "mycluster"
pd.cores    = 16
pd.runtime  = 5 # minutes

pilot = pm.submit_pilots(pd)
```

get_resource_config(*resource*, *schema=None*)
Returns a dictionary of the requested resource config

Security Contexts

class radical.pilot.**Context**(*ctype*, *thedict=None*)

__init__(*ctype*, *thedict=None*)
ctype: string ret: None

classmethod from_dict(*thedict*)
Creates a new object instance from a string. c._from_dict(x.as_dict) == x

1.6.2 Pilots and PilotManagers

PilotManagers

class radical.pilot.**PilotManager**(*session*)

A PilotManager manages *radical.pilot.ComputePilot* (page 46) instances that are submitted via the *radical.pilot.PilotManager.submit_pilots()* (page 44) method.

It is possible to attach one or more *Using Local and Remote HPC Resources* (page 61) to a PilotManager to outsource machine specific configuration parameters to an external configuration file.

Example:

```
s = radical.pilot.Session(database_url=DBURL)

pm = radical.pilot.PilotManager(session=s)

pd = radical.pilot.ComputePilotDescription()
pd.resource = "futuregrid.alamo"
pd.cpus = 16

p1 = pm.submit_pilots(pd)  # create first  pilot with 16 cores
p2 = pm.submit_pilots(pd)  # create second pilot with 16 cores

# Create a workload of 128 '/bin/sleep' compute units
compute_units = []
for unit_count in range(0, 128):
    cu = radical.pilot.ComputeUnitDescription()
    cu.executable = "/bin/sleep"
    cu.arguments = ['60']
    compute_units.append(cu)

# Combine the two pilots, the workload and a scheduler via
# a UnitManager.
um = radical.pilot.UnitManager(session=session,
                               scheduler=radical.pilot.SCHEDULER_ROUND_ROBIN)
um.add_pilot(p1)
um.submit_units(compute_units)
```

The pilot manager can issue notification on pilot state changes. Whenever state notification arrives, any callback registered for that notification is fired.

NOTE: State notifications can arrive out of order wrt the pilot state model!

__init__(*session*)

Creates a new PilotManager and attaches is to the session.

Arguments:

- session [*radical.pilot.Session* (page 40)]: The session instance to use.

Returns:

- A new *PilotManager* object [*radical.pilot.PilotManager* (page 43)].

close(*terminate=True*)

Shuts down the PilotManager.

Arguments:

- **terminate** [*bool*]: cancel non-final pilots if True (default)

is_valid(*term=True*)
Just as the Process' *is_valid()* call, we make sure that the component is still viable, and will raise an exception if not. Additionally to the health of the component's child process, we also check health of any sub-components and communication bridges.

as_dict()
Returns a dictionary representation of the PilotManager object.

uid
Returns the unique id.

list_pilots()
Returns the UIDs of the `radical.pilot.ComputePilots` managed by this pilot manager.

Returns:

- A list of *radical.pilot.ComputePilot* (page 46) UIDs [*string*].

submit_pilots(*descriptions*)
Submits on or more *radical.pilot.ComputePilot* (page 46) instances to the pilot manager.

Arguments:

- **descriptions** [*radical.pilot.ComputePilotDescription* (page 45) or list of *radical.pilot.ComputePilotDescription* (page 45)]: The description of the compute pilot instance(s) to create.

Returns:

- A list of *radical.pilot.ComputePilot* (page 46) objects.

get_pilots(*uids=None*)
Returns one or more compute pilots identified by their IDs.

Arguments:

- **uids** [*string* or *list of strings*]: The IDs of the compute pilot objects to return.

Returns:

- A list of *radical.pilot.ComputePilot* (page 46) objects.

wait_pilots(*uids=None*, *state=None*, *timeout=None*)
Returns when one or more `radical.pilot.ComputePilots` reach a specific state.

If *pilot_uids* is *None*, *wait_pilots* returns when **all** ComputePilots reach the state defined in *state*. This may include pilots which have previously terminated or waited upon.

Example:

```
# TODO -- add example
```

Arguments:

- **pilot_uids** [*string* or *list of strings*] If pilot_uids is set, only the ComputePilots with the specified uids are considered. If pilot_uids is *None* (default), all ComputePilots are considered.
- **state** [*string*] The state that ComputePilots have to reach in order for the call to return.

  By default *wait_pilots* waits for the ComputePilots to reach a terminal state, which can be one of the following:

  - `radical.pilot.rps.DONE`
  - `radical.pilot.rps.FAILED`

- radical.pilot.rps.CANCELED

- **timeout** [*float*] Timeout in seconds before the call returns regardless of Pilot state changes. The default value **None** waits forever.

cancel_pilots(*uids=None*, *_timeout=None*)

Cancel one or more radical.pilot.ComputePilots.

Arguments:

- **uids** [*string* or *list of strings*]: The IDs of the compute pilot objects to cancel.

register_callback(*cb*, *metric='PILOT_STATE'*, *cb_data=None*)

Registers a new callback function with the PilotManager. Manager-level callbacks get called if the specified metric changes. The default metric *PILOT_STATE* fires the callback if any of the ComputePilots managed by the PilotManager change their state.

All callback functions need to have the same signature:

```
def cb(obj, value, cb_data)
```

where object is a handle to the object that triggered the callback, value is the metric, and data is the data provided on callback registration.. In the example of *PILOT_STATE* above, the object would be the pilot in question, and the value would be the new state of the pilot.

Available metrics are:

- *PILOT_STATE*: fires when the state of any of the pilots which are managed by this pilot manager instance is changing. It communicates the pilot object instance and the pilots new state.

ComputePilotDescription

class radical.pilot.**ComputePilotDescription**(*from_dict=None*)

A ComputePilotDescription object describes the requirements and properties of a radical.pilot.Pilot and is passed as a parameter to *radical.pilot.PilotManager.submit_pilots()* (page 44) to instantiate and run a new pilot.

Note: A ComputePilotDescription **MUST** define at least *resource* (page 45), *cores* (page 46) and *runtime* (page 46).

Example:

```
pm = radical.pilot.PilotManager(session=s)

pd = radical.pilot.ComputePilotDescription()
pd.resource = "local.localhost"  # defined in futuregrid.json
pd.cores    = 16
pd.runtime  = 5 # minutes

pilot = pm.submit_pilots(pd)
```

resource

[Type: *string*] [**'mandatory'**] The key of a *Using Local and Remote HPC Resources* (page 61) entry. If the key exists, the machine-specifc configuration is loaded from the configuration once the ComputePilotDescription is passed to *radical.pilot.PilotManager.submit_pilots()* (page 44). If the key doesn't exist, a radical.pilot.pilotException is thrown.

access_schema
[Type: *string*] [**'optional'**] The key of an access mechanism to use. The valid access mechanism are defined in the resource configurations, see *Using Local and Remote HPC Resources* (page 61). The first one defined there is used by default, if no other is specified.

runtime
[Type: *int*] [**mandatory**] The maximum run time (wall-clock time) in **minutes** of the ComputePilot.

sandbox
[Type: *string*] [optional] The working ("sandbox") directory of the ComputePilot agent. This parameter is optional. If not set, it defaults to *radical.pilot.sandox* in your home or login directory.

> **Warning:** If you define a ComputePilot on an HPC cluster and you want to set *sandbox* manually, make sure that it points to a directory on a shared filesystem that can be reached from all compute nodes.

cores
[Type: *int*] [**mandatory**] The number of cores the pilot should allocate on the target resource.

NOTE: for local pilots, you can set a number larger than the physical machine limit when setting *RADICAL_PILOT_PROFILE* in your environment.

memory
[Type: *int*] [**optional**] The amount of memorty (in MB) the pilot should allocate on the target resource.

queue
[Type: *string*] [optional] The name of the job queue the pilot should get submitted to . If *queue* is defined in the resource configuration (`resource` (page 45)) defining *queue* will override it explicitly.

project
[Type: *string*] [optional] The name of the project / allocation to charge for used CPU time. If *project* is defined in the machine configuration (`resource` (page 45)), defining *project* will override it explicitly.

candidate_hosts
[Type: *list*] [optional] The list of names of hosts where this pilot is allowed to start on.

cleanup
[Type: *bool*] [optional] If cleanup is set to True, the pilot will delete its entire sandbox upon termination. This includes individual ComputeUnit sandboxes and all generated output data. Only log files will remain in the sandbox directory.

Pilots

class radical.pilot.**ComputePilot**(*pmgr*, *descr*)
A ComputePilot represent a resource overlay on a local or remote resource.

Note: A ComputePilot cannot be created directly. The factory method `radical.pilot.PilotManager.submit_pilots()` (page 44) has to be used instead.

Example:

```
pm = radical.pilot.PilotManager(session=s)

pd = radical.pilot.ComputePilotDescription()
pd.resource = "local.localhost"
pd.cores    = 2
```

(continues on next page)

(continued from previous page)

```
pd.runtime  = 5 # minutes

pilot = pm.submit_pilots(pd)
```

as_dict()
Returns a Python dictionary representation of the object.

session
Returns the pilot's session.

Returns:

- A *Session* (page 40).

pmgr
Returns the pilot's manager.

Returns:

- A *PilotManager* (page 43).

resource_details
Returns agent level resource information

uid
Returns the pilot's unique identifier.

The uid identifies the pilot within a *PilotManager* (page 43).

Returns:

- A unique identifier (string).

state
Returns the current state of the pilot.

Returns:

- state (string enum)

log
Returns a list of human readable [timestamp, string] tuples describing various events during the pilot's lifetime. Those strings are not normative, only informative!

Returns:

- log (list of [timestamp, string] tuples)

stdout
Returns a snapshot of the pilot's STDOUT stream.

If this property is queried before the pilot has reached 'DONE' or 'FAILED' state it will return None.

Returns:

- stdout (string)

stderr
Returns a snapshot of the pilot's STDERR stream.

If this property is queried before the pilot has reached 'DONE' or 'FAILED' state it will return None.

Returns:

- stderr (string)

resource
Returns the resource tag of this pilot.

Returns:

- A resource tag (string)

pilot_sandbox
Returns the full sandbox URL of this pilot, if that is already known, or 'None' otherwise.

Returns:

- A string

description
Returns the description the pilot was started with, as a dictionary.

Returns:

- description (dict)

register_callback(*cb*, *metric='PILOT_STATE'*, *cb_data=None*)
Registers a callback function that is triggered every time the pilot's state changes.

All callback functions need to have the same signature:

```
def cb(obj, state)
```

where `object` is a handle to the object that triggered the callback and `state` is the new state of that object. If 'cb_data' is given, then the 'cb' signature changes to

def cb(obj, state, cb_data)

and 'cb_data' are passed along.

wait(*state=None*, *timeout=None*)
Returns when the pilot reaches a specific state or when an optional timeout is reached.

Arguments:

- **state** [*list of strings*] The state(s) that pilot has to reach in order for the call to return.

  By default *wait* waits for the pilot to reach a **final** state, which can be one of the following:

  - `radical.pilot.states.DONE`
  - `radical.pilot.states.FAILED`
  - `radical.pilot.states.CANCELED`

- **timeout** [*float*] Optional timeout in seconds before the call returns regardless whether the pilot has reached the desired state or not. The default value **None** never times out.

cancel()
Cancel the pilot.

stage_in(*directives*)
Stages the content of the staging directive into the pilot's staging area

1.6.3 ComputeUnits and UnitManagers

UnitManager

class radical.pilot.**UnitManager**(*session*, *scheduler=None*)

A UnitManager manages *radical.pilot.ComputeUnit* (page 55) instances which represent the **executable** workload in RADICAL-Pilot. A UnitManager connects the ComputeUnits with one or more `Pilot` instances (which represent the workload **executors** in RADICAL-Pilot) and a **scheduler** which determines which *ComputeUnit* (page 55) gets executed on which `Pilot`.

Example:

```
s = rp.Session(database_url=DBURL)

pm = rp.PilotManager(session=s)

pd = rp.ComputePilotDescription()
pd.resource = "futuregrid.alamo"
pd.cores = 16

p1 = pm.submit_pilots(pd) # create first pilot with 16 cores
p2 = pm.submit_pilots(pd) # create second pilot with 16 cores

# Create a workload of 128 '/bin/sleep' compute units
compute_units = []
for unit_count in range(0, 128):
    cu = rp.ComputeUnitDescription()
    cu.executable = "/bin/sleep"
    cu.arguments = ['60']
    compute_units.append(cu)

# Combine the two pilots, the workload and a scheduler via
# a UnitManager.
um = rp.UnitManager(session=session,
                    scheduler=rp.SCHEDULER_ROUND_ROBIN)
um.add_pilot(p1)
um.submit_units(compute_units)
```

The unit manager can issue notification on unit state changes. Whenever state notification arrives, any callback registered for that notification is fired.

NOTE: State notifications can arrive out of order wrt the unit state model!

__init__(*session*, *scheduler=None*)

Creates a new UnitManager and attaches it to the session.

Arguments:

- session [*radical.pilot.Session* (page 40)]: The session instance to use.
- scheduler (*string*): The name of the scheduler plug-in to use.

Returns:

- A new *UnitManager* object [*radical.pilot.UnitManager* (page 49)].

close()

Shut down the UnitManager, and all umgr components.

is_valid(*term=True*)

Just as the Process' *is_valid()* call, we make sure that the component is still viable, and will raise an

exception if not. Additionally to the health of the component's child process, we also check health of any sub-components and communication bridges.

as_dict()
Returns a dictionary representation of the UnitManager object.

uid
Returns the unique id.

scheduler
Returns the scheduler name.

add_pilots(*pilots*)
Associates one or more pilots with the unit manager.

Arguments:

- **pilots** [*radical.pilot.ComputePilot* (page 46) or list of *radical.pilot.ComputePilot* (page 46)]: The pilot objects that will be added to the unit manager.

list_pilots()
Lists the UIDs of the pilots currently associated with the unit manager.

Returns:

- A list of *radical.pilot.ComputePilot* (page 46) UIDs [*string*].

get_pilots()
Get the pilots instances currently associated with the unit manager.

Returns:

- A list of *radical.pilot.ComputePilot* (page 46) instances.

remove_pilots(*pilot_ids*, *drain=False*)
Disassociates one or more pilots from the unit manager.

After a pilot has been removed from a unit manager, it won't process any of the unit manager's units anymore. Calling *remove_pilots* doesn't stop the pilot itself.

Arguments:

- **drain** [*boolean*]: Drain determines what happens to the units which are managed by the removed pilot(s). If *True*, all units currently assigned to the pilot are allowed to finish execution. If *False* (the default), then non-final units will be canceled.

list_units()
Returns the UIDs of the *radical.pilot.ComputeUnit* (page 55) managed by this unit manager.

Returns:

- A list of *radical.pilot.ComputeUnit* (page 55) UIDs [*string*].

submit_units(*descriptions*)
Submits on or more *radical.pilot.ComputeUnit* (page 55) instances to the unit manager.

Arguments:

- **descriptions** [*radical.pilot.ComputeUnitDescription* (page 52) or list of *radical.pilot.ComputeUnitDescription* (page 52)]: The description of the compute unit instance(s) to create.

Returns:

- A list of *radical.pilot.ComputeUnit* (page 55) objects.

get_units(*uids=None*)
Returns one or more compute units identified by their IDs.

Arguments:

- **uids** [*string* or *list of strings*]: The IDs of the compute unit objects to return.

Returns:

- A list of *radical.pilot.ComputeUnit* (page 55) objects.

wait_units(*uids=None*, *state=None*, *timeout=None*)
Returns when one or more `radical.pilot.ComputeUnits` reach a specific state.

If *uids* is *None*, *wait_units* returns when **all** ComputeUnits reach the state defined in *state*. This may include units which have previously terminated or waited upon.

Example:

```
# TODO -- add example
```

Arguments:

- **uids** [*string* or *list of strings*] If uids is set, only the ComputeUnits with the specified uids are considered. If uids is *None* (default), all ComputeUnits are considered.
- **state** [*string*] The state that ComputeUnits have to reach in order for the call to return.

  By default *wait_units* waits for the ComputeUnits to reach a terminal state, which can be one of the following:

  - `radical.pilot.rps.DONE`
  - `radical.pilot.rps.FAILED`
  - `radical.pilot.rps.CANCELED`
- **timeout** [*float*] Timeout in seconds before the call returns regardless of Pilot state changes. The default value **None** waits forever.

cancel_units(*uids=None*)
Cancel one or more `radical.pilot.ComputeUnits`.

Note that cancellation of units is *immediate*, i.e. their state is immediately set to *CANCELED*, even if some RP component may still operate on the units. Specifically, other state transitions, including other final states (*DONE*, *FAILED*) can occur *after* cancellation. This is a side effect of an optimization: we consider this acceptable tradeoff in the sense "Oh, that unit was DONE at point of cancellation – ok, we can use the results, sure!".

If that behavior is not wanted, set the environment variable:

export RADICAL_PILOT_STRICT_CANCEL=True

Arguments:

- **uids** [*string* or *list of strings*]: The IDs of the compute units objects to cancel.

register_callback(*cb*, *metric='UNIT_STATE'*, *cb_data=None*)
Registers a new callback function with the UnitManager. Manager-level callbacks get called if the specified metric changes. The default metric *UNIT_STATE* fires the callback if any of the ComputeUnits managed by the PilotManager change their state.

All callback functions need to have the same signature:

```
def cb(obj, value, cb_data)
```

where `object` is a handle to the object that triggered the callback, `value` is the metric, and `data` is the data provided on callback registration.. In the example of *UNIT_STATE* above, the object would be the unit in question, and the value would be the new state of the unit.

Available metrics are:

- *UNIT_STATE*: fires when the state of any of the units which are managed by this unit manager instance is changing. It communicates the unit object instance and the units new state.
- *WAIT_QUEUE_SIZE*: fires when the number of unscheduled units (i.e. of units which have not been assigned to a pilot for execution) changes.

ComputeUnitDescription

class radical.pilot.**ComputeUnitDescription**(*from_dict=None*)

A ComputeUnitDescription object describes the requirements and properties of a *radical.pilot.ComputeUnit* (page 55) and is passed as a parameter to *radical.pilot.UnitManager.submit_units()* (page 50) to instantiate and run a new unit.

Note: A ComputeUnitDescription **MUST** define at least an *executable* or *kernel* – all other elements are optional.

Example:

```
# TODO
```

executable

The executable to launch (*string*). The executable is expected to be either available via *$PATH* on the target resource, or to be an absolute path.

default: *None*

cpu_processes
number of application processes to start on CPU cores
default: 0

cpu_threads
number of threads each process will start on CPU cores
default: 1

cpu_process_type
process type, determines startup method (POSIX, MPI)
default: POSIX

cpu_thread_type
thread type, influences startup and environment (POSIX, OpenMP)
default: POSIX

gpu_processes
number of application processes to start on GPU cores
default: 0

gpu_threads
number of threads each process will start on GPU cores
default: 1

gpu_process_type
process type, determines startup method (POSIX, MPI)
default: POSIX

gpu_thread_type
thread type, influences startup and environment (POSIX, OpenMP, CUDA)
default: POSIX

lfs (*local file storage*)
amount of data (MB) required on the local file system of the node
default: 0

name
A descriptive name for the compute unit (*string*). This attribute can be used to map individual units back to application level workloads.

default: *None*

arguments
The command line arguments for the given *executable* (*list* of *strings*).

default: *[]*

environment
Environment variables to set in the environment before execution (*dict*).

default: *{}*

stdout
The name of the file to store stdout in (*string*).

default: *STDOUT*

stderr
The name of the file to store stderr in (*string*).

default: *STDERR*

input_staging
The files that need to be staged before execution (*list* of *staging directives*, see below).

default: *{}*

output_staging
The files that need to be staged after execution (*list* of *staging directives*, see below).

default: *{}*

pre_exec
Actions (shell commands) to perform before this task starts (*list* of *strings*). Note that the set of shell commands given here are expected to load environments, check for work directories and data, etc. They are not expected to consume any significant amount of CPU time or other resources! Deviating from that rule will likely result in reduced overall throughput.

No assumption should be made as to where these commands are executed (although RP attempts to perform them in the unit's execution environment).

No assumption should be made on the specific shell environment the commands are executed in.

Errors in executing these commands will result in the unit to enter *FAILED* state, and no execution of the actual workload will be attempted.

default: *[]*

post_exec
Actions (shell commands) to perform after this task finishes (*list* of *strings*). The same remarks as on *pre_exec* apply, inclusive the point on error handling, which again will cause the unit to fail, even if the actual execution was successful..

default: *[]*

kernel
Name of a simulation kernel which expands to description attributes once the unit is scheduled to a pilot (and resource).

Note: TODO: explain in detail, reference ENMDTK.

---

default: *None*

restartable
If the unit starts to execute on a pilot, but cannot finish because the pilot fails or is canceled, can the unit be restarted on a different pilot / resource?

default: *False*

metadata
user defined metadata

default: *None*

cleanup
If cleanup (a *bool*) is set to *True*, the pilot will delete the entire unit sandbox upon termination. This includes all generated output data in that sandbox. Output staging will be performed before cleanup.

Note that unit sandboxes are also deleted if the pilot's own *cleanup* flag is set.

default: *False*

pilot
If specified as *string* (pilot uid), the unit is submitted to the pilot with the given ID. If that pilot is not known to the unit manager, an exception is raised.

The Staging Directives are specified using a dict in the following form:

staging_directive = { 'source' : None, # see 'Location' below 'target' : None, # see 'Location' below 'action' : None, # See 'Action operators' below 'flags' : None, # See 'Flags' below 'priority': 0 # Control ordering of actions (unused)

}

source and *target* locations can be given as strings or *ru.URL* instances. Strings containing *://* are converted into URLs immediately. Otherwise they are considered absolute or relative paths and are then interpreted in the context of the client's working directory.

RP accepts the following special URL schemas:

- *client://* : relative to the client's working directory
- *resource://*: relative to the RP sandbox on the target resource
- *pilot://* : relative to the pilot sandbox on the target resource
- *unit://* : relative to the unit sandbox on the target resource

In all these cases, the *hostname* element of the URL is expected to be empty, and the path is *always* considered relative to the locations specified above (even though URLs usually don't have a notion of relative paths).

RP accepts the following action operators:

- rp.TRANSFER: remote file transfer from *source* URL to *target* URL.
- rp.COPY : local file copy, ie. not crossing host boundaries
- rp.MOVE : local file move
- rp.LINK : local file symlink

rp.CREATE_PARENTS: create the directory hierarchy for targets on the fly rp.RECURSIVE : if *source* is a directory, handle it recursively

verify()
Verify that the description is syntactically and semantically correct. This method encapsulates checks beyond the SAGA attribute level checks.

ComputeUnit

class radical.pilot.**ComputeUnit**(*umgr*, *descr*)
A ComputeUnit represent a 'task' that is executed on a ComputePilot. ComputeUnits allow to control and query the state of this task.

Note: A unit cannot be created directly. The factory method *radical.pilot.UnitManager.submit_units()* (page 50) has to be used instead.

Example:

```
umgr = radical.pilot.UnitManager(session=s)

ud = radical.pilot.ComputeUnitDescription()
ud.executable = "/bin/date"

unit = umgr.submit_units(ud)
```

---

as_dict()
Returns a Python dictionary representation of the object.

session
Returns the unit's session.

Returns:

- A *Session* (page 40).

umgr
Returns the unit's manager.

Returns:

- A *UnitManager* (page 49).

uid
Returns the unit's unique identifier.

The uid identifies the unit within a *UnitManager* (page 49).

Returns:

- A unique identifier (string).

name
Returns the unit's application specified name.

Returns:

- A name (string).

state
Returns the current state of the unit.

Returns:

- state (string enum)

exit_code
Returns the exit code of the unit, if that is already known, or 'None' otherwise.

Returns:

- exit code (int)

stdout
Returns a snapshot of the executable's STDOUT stream.

If this property is queried before the unit has reached 'DONE' or 'FAILED' state it will return None.

Returns:

- stdout (string)

stderr
Returns a snapshot of the executable's STDERR stream.

If this property is queried before the unit has reached 'DONE' or 'FAILED' state it will return None.

Returns:

- stderr (string)

pilot
Returns the pilot ID of this unit, if that is already known, or 'None' otherwise.

Returns:

- A pilot ID (string)

unit_sandbox
Returns the full sandbox URL of this unit, if that is already known, or 'None' otherwise.

Returns:

- A URL (radical.utils.Url).

description
Returns the description the unit was started with, as a dictionary.

Returns:

- description (dict)

metadata
Returns the metadata field of the unit's description

register_callback(*cb*, *cb_data=None*)
Registers a callback function that is triggered every time the unit's state changes.

All callback functions need to have the same signature:

```
def cb(obj, state)
```

where `object` is a handle to the object that triggered the callback and `state` is the new state of that object. If 'cb_data' is given, then the 'cb' signature changes to

def cb(obj, state, cb_data)

and 'cb_data' are passed along.

wait(*state=None*, *timeout=None*)
Returns when the unit reaches a specific state or when an optional timeout is reached.

Arguments:

- **state** [*list of strings*] The state(s) that unit has to reach in order for the call to return.

  By default *wait* waits for the unit to reach a **final** state, which can be one of the following:

  - `radical.pilot.states.DONE`
  - `radical.pilot.states.FAILED`
  - `radical.pilot.states.CANCELED`

- **timeout** [*float*] Optional timeout in seconds before the call returns regardless whether the unit has reached the desired state or not. The default value **None** never times out.

cancel()
Cancel the unit.

1.6.4 Exceptions

class radical.pilot.**PilotException**(*msg*, *obj=None*)

Parameters

- **msg** (*string*) – Error message, indicating the cause for the exception being raised.
- **obj** (*object*) – RADICAL-Pilot object on whose activity the exception was raised.

Raises –

The base class for all RADICAL-Pilot Exception classes – this exception type is never raised directly, but can be used to catch all RADICAL-Pilot exceptions within a single *except* clause.

The exception message and originating object are also accessable as class attributes (`e.object()` and `e.message()`). The `__str__()` operator redirects to *get_message()* (page 57).

get_object()
Return the object instance on whose activity the exception was raised.

get_message()
Return the error message associated with the exception

class radical.pilot.**DatabaseError**(*msg*, *obj=None*)
TODO: Document me!

1.7 Data Staging

Note: Currently RADICAL-Pilot only supports data on file abstraction level, so *data* == *files* at this moment.

Many, if not all, programs require input data to operate and create output data as a result in some form or shape. RADICAL-Pilot has a set of constructs that allows the user to specify the required staging of input and output files for a Compute Unit.

The primary constructs are on the level of the Compute Unit (Description) which are discussed in the next section. For more elaborate use-cases we also have constructs on the Compute Pilot level, which are discussed later in this chapter.

Note: RADICAL-Pilot uses system calls for local file operations and SAGA for remote transfers and URL specification.

1.7.1 Compute Unit I/O

To instruct RADICAL-Pilot to handle files for you, there are two things to take care of. First you need to specify the respective input and output files for the Compute Unit in so called *staging directives*. Additionally you need to associate these *staging directives* to the the Compute Unit by means of the `input_staging` and `output_staging` members.

What it looks like

The following code snippet shows this in action:

```
INPUT_FILE_NAME  = "INPUT_FILE.TXT"
OUTPUT_FILE_NAME = "OUTPUT_FILE.TXT"

# This executes: "/usr/bin/sort -o OUTPUT_FILE.TXT INPUT_FILE.TXT"
cud = radical.pilot.ComputeUnitDescription()
cud.executable = "/usr/bin/sort"
cud.arguments = ["-o", OUTPUT_FILE_NAME, INPUT_FILE_NAME]
cud.input_staging  = INPUT_FILE_NAME
cud.output_staging = OUTPUT_FILE_NAME
```

Here the *staging directives* `INPUT_FILE_NAME` and `OUTPUT_FILE_NAME` are simple strings that both specify a single filename and are associated to the Compute Unit Description `cud` for input and output respectively.

What this does is that the file *INPUT_FILE.TXT* is transferred from the local directory to the directory where the task is executed. After the task has run, the file *OUTPUT_FILE.TXT* that has been created by the task, will be transferred back to the local directory.

The *String-Based Input and Output Transfer* (page 60) example demonstrates this in full glory.

Staging Directives

The format of the *staging directives* can either be a string as above or a dict of the following structure:

```
staging_directive = {
    'source':   source,    # radical.pilot.Url() or string (MANDATORY).
    'target':   target,    # radical.pilot.Url() or string (OPTIONAL).
    'action':   action,    # One of COPY, LINK, MOVE, TRANSFER or TARBALL (OPTIONAL).
    'flags':    flags,     # Zero or more of CREATE_PARENTS or SKIP_FAILED (OPTIONAL).
    'priority': priority   # A number to instruct ordering (OPTIONAL).
}
```

The semantics of the keys from the dict are as follows:

- **`source` (default: None) and `target` (default: os.path.basename(source)):** In case of the *staging directive* being used for *input*, then the `source` refers to the location to get the input files from, e.g. the local working directory on your laptop or a remote data repository, and `target` refers to the working directory of the ComputeUnit. Alternatively, in case of the *staging directive* being used for *output*, then the `source` refers to the output files being generated by the ComputeUnit in the working directory and `target` refers to the location where you need to store the output data, e.g. back to your laptop or some remote data repository.
- **`action` (default: TRANSFER):** The *ultimate* goal is to make data available to the application kernel in the ComputeUnit and to be able to make the results available for further use. Depending on the relative location of the working directory of the `source` to the `target` location, the action can be `COPY` (local resource), `LINK` (same file system), `MOVE` (local resource), `TRANSFER` (to a remote resource), or `TARBALL` (transfer to a remote resource after tarring files).
- **`flags` (default: [CREATE_PARENTS, SKIP_FAILED]):** By passing certain flags we can influence the behavior of the action. Available flags are:
  - `CREATE_PARENTS`: Create parent directories while writing file.
  - `SKIP_FAILED`: Don't stage out files if tasks failed.

  In case of multiple values these can be passed as a list.
- **`priority` (default: 0):** This optional field can be used to instruct the backend to priority the actions on the `staging directives`. E.g. to first stage the output that is required for immediate further analysis and afterwards some output files that are of secondary concern.

The *Dictionary-Based Input and Output Transfer* (page 61) example demonstrates this in full glory.

When the *staging directives* are specified as a string as we did earlier, that implies a *staging directive* where the `source` and the `target` are equal to the content of the string, the `action` is set to the default action `TRANSFER`, the `flags` are set to the default flags `CREATE_PARENTS` and `SKIP_FAILED`, and the `priority` is set to the default value `0`:

```
'INPUT_FILE.TXT' == {
    'source':   'INPUT_FILE.TXT',
    'target':   'INPUT_FILE.TXT',
    'action':   TRANSFER,
    'flags':    [CREATE_PARENTS, SKIP_FAILED],
    'priority': 0
}
```

Staging Area

As the pilot job creates an abstraction for a computational resource, the user does not necessarily know where the working directory of the Compute Pilot or the Compute Unit is. Even if he knows, the user might not have direct access to it. For this situation we have the staging area, which is a special construct so that the user can specify files

relative to or in the working directory without knowing the exact location. This can be done using the following URL format:

```
'staging:///INPUT_FILE.TXT'
```

The *Pipeline* (page 61) example demonstrates this in full glory.

1.7.2 Compute Pilot I/O

As mentioned earlier, in addition to the constructs on Compute Unit-level RADICAL-Pilot also has constructs on Compute Pilot-level. The main rationale for this is that often there is (input) data to be shared between multiple Compute Units. Instead of transferring the same files for every Compute Unit, we can transfer the data once to the Pilot, and then make it available to every Compute Unit that needs it.

This works in a similar way as the Compute Unit-IO, where we use also use the Staging Directive to specify the I/O transaction The difference is that in this case, the Staging Directive is not associated to the Description, but used in a direct method call `pilot.stage_in(sd_pilot)`.

```
# Configure the staging directive for to insert the shared file into
# the pilot staging directory.
sd_pilot = {'source': shared_input_file_url,
            'target': os.path.join(MY_STAGING_AREA, SHARED_INPUT_FILE),
            'action': radical.pilot.TRANSFER
}
# Synchronously stage the data to the pilot
pilot.stage_in(sd_pilot)
```

The *Shared Input Files* (page 61) example demonstrates this in full glory.

Note: The call to `stage_in()` is synchronous and will return once the transfer is complete.

1.7.3 Examples

Note: All of the following examples are configured to run on localhost, but they can be easily changed to run on a remote resource by modifying the resource specification in the Compute Pilot Description. Also note the comments in *Staging Area* (page 59) when changing the examples to a remote target.

These examples require an installation of RADICAL-Pilot of course. There are download links for each of the examples.

String-Based Input and Output Transfer

This example demonstrates the simplest form of the data staging capabilities. The example demonstrates how a local input file is staged through RADICAL-Pilot, processed by the Compute Unit and the resulting output file is staged back to the local environment.

Note: Download the example:

```
curl -O https://raw.githubusercontent.com/radical-cybertools/radical.pilot/
readthedocs/examples/io_staging_simple.py
```

Dictionary-Based Input and Output Transfer

This example demonstrates the use of the staging directives structure to have more control over the staging behavior. The flow of the example is similar to that of the previous example, but here we show that by using the dict-based Staging Directive, one can specify different names and paths for the local and remote files, a feature that is often required in real-world applications.

Note: Download the example:

```
curl -O https://raw.githubusercontent.com/radical-cybertools/radical.pilot/
readthedocs/examples/io_staging_dict.py
```

Shared Input Files

This example demonstrates the staging of a shared input file by means of the stage_in() method of the pilot and consequently making that available to all compute units.

Note: Download the example:

```
curl -O https://raw.githubusercontent.com/radical-cybertools/radical.pilot/
readthedocs/examples/io_staging_shared.py
```

Pipeline

This example demonstrates a two-step pipeline that makes use of a remote pilot staging area, where the first step of the pipeline copies the intermediate output into and that is picked up by the second step in the pipeline.

Note: Download the example:

```
curl -O https://raw.githubusercontent.com/radical-cybertools/radical.pilot/
readthedocs/examples/io_staging_pipeline.py
```

1.8 Using Local and Remote HPC Resources

1.8.1 Introduction

RADICAL-Pilot allows you to launch a ComputePilot allocating a large number of cores on a remote HPC cluster. The ComputePilot is then used to run multiple ComputeUnits with small core-counts. This separates resource allocation and management from resource usage, and avoids HPC cluster queue policies and waiting times which can significantly reduce the total time to completion of your application.

If you want to use a remote HPC resource (in this example a cluster named "Archer", located at EPSRC, UK) you have to define it in the ComputePilotDescription like this:

```
pdesc = radical.pilot.ComputePilotDescription()
pdesc.resource = "epsrc.archer"
pdesc.project  = "e1234"
pdesc.runtime  = 60
pdesc.cores    = 128
```

Using a `resource` key other than "local.localhost" implicitly tells RADICAL-Pilot that it is targeting a remote resource. RADICAL-Pilot is using the SSH/GSISSH (and SFTP/GSISFTP) protocols to communicate with remote resources. The next section, *Configuring SSH Access* (page 62) provides some details about SSH set-up. *Pre-Configured Resources* (page 62) lists the resource keys that are already defined and ready to be used in RADICAL-Pilot.

1.8.2 Configuring SSH Access

If you can manually SSH into the target resource, RADICAL-Pilot can do the same. While RADICAl-Pilot supports username and password authentication, it is highly-advisable to set-up password-less ssh keys for the resource you want to use. If you are not familiar with how to setup password-less ssh keys, check out this link.

All SSH-specific informations, like remote usernames, passwords, and keyfiles, are set in a `Context` object. For example, if you want to tell RADICAL-Pilot your user-id on the remote resource, use the following construct:

```
session = radical.pilot.Session()

c = radical.pilot.Context('ssh')
c.user_id = "tg802352"
session.add_context(c)
```

Note: **Tip:** You can create an empty file called *.hushlogin* in your home directory to turn off the system messages you see on your screen at every login. This can help if you encounter random connection problems with RADICAL-Pilot.

1.8.3 Pre-Configured Resources

Resource configurations are a set of key/value dictionaries with details of a remote resource like queuing-, file-system-, and environment-. Once a configuration file is available for a given resource, a user chooses that pre-configured resource in her code like this:

```
pdesc = radical.pilot.ComputePilotDescription()
pdesc.resource   = "epsrc.archer"
pdesc.project    = "e1234"
pdesc.runtime    = 60
pdesc.cores      = 128
pdesc.queue      = "large"
```

The RADICAL-Pilot developer team maintains a growing set of resource configuration files. Several of the settings included there can be overridden in the `ComputePilotDescription` object. For example, the snipped above replaces the default queue `standard` with the queue `large`. For a list of supported configurations, see chapter_resources - those resource files live under `radical/pilot/configs/`.

1.8.4 Writing a Custom Resource Configuration File

If you want to use RADICAL-Pilot with a resource that is not in any of the provided resource configuration files, you can write your own, and drop it in `$HOME/.radical/pilot/configs/`

`<your_resource_configuration_file_name>.json`.

Note: Be advised that you may need specific knowledge about the target resource to do so. Also, while RADICAL-Pilot can handle very different types of systems and batch system, it may run into trouble on specific configurations or software versions we did not encounter before. If you run into trouble using a resource not in our list of officially supported ones, please drop us a note on the RADICAL-Pilot users mailing list.

A configuration file has to be valid JSON. The structure is as follows:

```
# filename: lrz.json
{
    "supermuc":
    {
        "description"                 : "The SuperMUC petascale HPC cluster at LRZ.",
        "notes"                       : "Access only from registered IP addresses.",
        "schemas"                     : ["gsissh", "ssh"],
        "ssh"                         :
        {
            "job_manager_endpoint"    : "loadl+ssh://supermuc.lrz.de/",
            "filesystem_endpoint"     : "sftp://supermuc.lrz.de/"
        },
        "gsissh"                      :
        {
            "job_manager_endpoint"    : "loadl+gsissh://supermuc.lrz.de:2222/",
            "filesystem_endpoint"     : "gsisftp://supermuc.lrz.de:2222/"
        },
        "default_queue"               : "test",
        "lrms"                        : "LOADL",
        "task_launch_method"          : "SSH",
        "mpi_launch_method"           : "MPIEXEC",
        "forward_tunnel_endpoint"     : "login03",
        "global_virtenv"              : "/home/hpc/pr87be/di29sut/pilotve",
        "pre_bootstrap_0"             : ["source /etc/profile",
                                         "source /etc/profile.d/modules.sh",
                                         "module load python/2.7.6",
                                         "module unload mpi.ibm", "module load mpi.
↪intel",
                                         "source /home/hpc/pr87be/di29sut/pilotve/bin/
↪activate"
                                        ],
        "valid_roots"                 : ["/home", "/gpfs/work", "/gpfs/scratch"],
        "agent_type"                  : "multicore",
        "agent_scheduler"             : "CONTINUOUS",
        "agent_spawner"               : "POPEN",
        "pilot_agent"                 : "radical-pilot-agent-multicore.py",
        "pilot_dist"                  : "default"
    },
    "ANOTHER_KEY_NAME":
    {
        ...
    }
}
```

The name of your file (here `lrz.json`) together with the name of the resource (`supermuc`) form the resource key which is used in the *class:ComputePilotDescription* resource attribute (`lrz.supermuc`).

All fields are mandatory, unless indicated otherwise below.

- `description`: a human readable description of the resource.
- `notes`: information needed to form valid pilot descriptions, such as what parameter are required, etc.
- `schemas`: allowed values for the `access_schema` parameter of the pilot description. The first schema in the list is used by default. For each schema, a subsection is needed which specifies `job_manager_endpoint` and `filesystem_endpoint`.
- `job_manager_endpoint`: access url for pilot submission (interpreted by SAGA).
- `filesystem_endpoint`: access url for file staging (interpreted by SAGA).
- `default_queue`: queue to use for pilot submission (optional).
- `lrms`: type of job management system. Valid values are: `LOADL`, `LSF`, `PBSPRO`, `SGE`, `SLURM`, `TORQUE`, `FORK`.
- `task_launch_method`: type of compute node access, required for non-MPI units. Valid values are: `SSH`,''APRUN'' or `LOCAL`.
- `mpi_launch_method`: type of MPI support, required for MPI units. Valid values are: `MPIRUN`, `MPIEXEC`, `APRUN`, `IBRUN` or `POE`.
- `python_interpreter`: path to python (optional).
- `python_dist`: *anaconda* or *default*, ie. not *anaconda* (mandatory).
- `pre_bootstrap_0`: list of commands to execute for initialization of main agent (optional).
- `pre_bootstrap_1`: list of commands to execute for initialization of sub-agent (optional).
- `valid_roots`: list of shared file system roots (optional). Note: pilot sandboxes must lie under these roots.
- `pilot_agent`: type of pilot agent to use. Currently: `radical-pilot-agent-multicore.py`.
- `forward_tunnel_endpoint`: name of the host which can be used to create ssh tunnels from the compute nodes to the outside world (optional).

Several configuration files are part of the RADICAL-Pilot installation, and live under `radical/pilot/configs/` .

1.8.5 Customizing Resource Configurations Programatically

The set of resource configurations available to the RADICAL-Pilot session is accessible programmatically. The example below changes the `default_queue` for the `epsrc.archer` resource.

```
import radical.pilot as rp
import pprint

RESOURCE = 'epsrc.archer'

# get a pre-installed resource configuration
session = rp.Session()
cfg = session.get_resource_config(RESOURCE)
pprint.pprint (cfg)

# create a new config based on the old one, and set a different launch method
new_cfg = rp.ResourceConfig(RESOURCE, cfg)
new_cfg.default_queue = 'royal_treatment'

# now add the entry back.  As we did not change the config name, this will
# replace the original configuration.  A completely new configuration would
```

(continues on next page)

(continued from previous page)

```
# need a unique label.
session.add_resource_config(new_cfg)
pprint.pprint (session.get_resource_config(RESOURCE))
```

1.9 Unit Scheduler

1.9.1 Introduction

The *class:radical.pilot.UnitManager* dispatches compute units to available pilots for execution. It does so according to some schedulin algorithm, which can be selected when instantiating the manager. Momentarily we support two schduling algorithms: 'Round-Robin' and 'Backfilling'. New schedulers can be added to radical.pilot – please contact us on the mailing list 'radical-pilot-devel@googlegroups.com' for details on support.

Note that radical.pilot enacts a second scheduling step: once a pilot agent takes ownership of units which the unit manager scheduler assigned to it, the agent scheduler will place the units on the set of resources (cores) that agent is managing. The agent scheduler can be configured via agent and resource configuration files (see chapter_resources).

1.9.2 Round-Robin Scheduler (`SCHEDULER_ROUND_ROBIN`)

The Round-Robin scheduler will fairly distributed arriving compute units over the set of known pilots, independent of unit state, expected workload, pilot state or pilot lifetime. As such, it is a fairly simplistic, but also a very fast scheduler, which does not impose any additional communication roundtrips between the unit manager and pilot agents.

1.9.3 Backfilling Scheduler (`SCHEDULER_BACKFILLING`)

The backfilling scheduler does a better job at actual load balancing, but at the cost of additional communication roundtrips. It depends on the actual application workload if that load balancing is beneficial or not.

Backfilling is most beneficial for large numbers of pilots and for relatively long running units (where the CU runtime is significantly longer than the communication roundtrip time between unit manager and pilot agent).

In general we thus recomment to *not* use backfilling

- for a single pilot;
- for large numbers of short-running CUs.

The backfilling scheduler (BF) will only dispatch units to pilot agents once the pilot agent is in 'RUNNING' state. The units will thus get executed even if one of the pilots never reaches that state: the load will be distributed between pilots which become 'ACTIVE'.

The BF will only dispatch as many units to an agent which the agent can, in principle, execute concurrently. No units will be waiting in the agent's own scheduler queue. The BF will react on unit termination events, and will then backfill (!) the agent with any remaining units. The agent will remain under-utilized during that communication.

In order to minimize agent under-utilization, the user can set the environment variable *RADICAL_PILOT_BF_OVERSUBSCRIPTION*, which specifies (in percent) with how many units the BF can overload the pilot agent, without waiting for unit termination notices. This mechanism effectively hides the communication latencies, as long as unit runtimes are significantly larger than the communication delays. Thre default oversubscription value is '0%', i.e. no oversubscription.

1.10 Testing

1.10.1 Introduction

Along with RADICAL-Pilot's functionality, we develop a growing set of unit tests. The unit test source code can be found in `src/radical/pilot/tests`. You can run the unit tests directly from the source directory without haing to install RADICAL-Pilot first:

```
export RADICAL_PILOT_VERBOSE=debug
export RADICAL_PILOT_TEST_DBNAME=rbtest_`date | md5sum | cut -c 1-32`
python setup.py test
```

Note: `RADICAL_PILOT_TEST_DBNAME` creates a somewhat of a random database name for the tests. This prevents interference caused by tests run against the same MongoDB concurrently.

If you run the same command in an environment where RADICAL-Pilot is already installed, the unit tests will test the installed version instead of the source version.

1.10.2 Remote Testing

The RADICAL-Pilot unit tests use pilot agents launched on the local machine (*localhost*) by default. However, it is possible to run a subset of the unit tests (`src/radical/pilot/tests/remote/`) on a remote machine. Remote testing can be controlled via a set of environment variables:

Environment Variable	What
RADICAL_PILOT_TEST_REMOTE_RESOURCE	The name (key) of the resource.
RADICAL_PILOT_TEST_REMOTE_SSH_USER_ID	The user ID on the remote system.
RADICAL_PILOT_TEST_REMOTE_SSH_USER_KEY	The SSH key to use for the connection.
RADICAL_PILOT_TEST_REMOTE_WORKDIR	The working directory on the remote system.
RADICAL_PILOT_TEST_REMOTE_CORES	The number of cores to allocate.
RADICAL_PILOT_TEST_REMOTE_NUM_CUS	The number of Compute Units to run.
RADICAL_PILOT_TEST_TIMEOUT	Set a timeout in minutes after which the tests will terminate.

So if for example you want to run the unit tests on Futuregrid's _India_ cluster (http://manual.futuregrid.org/hardware.html), run

```
RADICAL_PILOT_VERBOSE=debug \
RADICAL_PILOT_TEST_REMOTE_SSH_USER_ID=oweidner # optional \
RADICAL_PILOT_TEST_REMOTE_RESOURCE=futuregrid.INDIA \
RADICAL_PILOT_TEST_REMOTE_WORKDIR=/N/u/oweidner/radical.pilot.sandbox \
RADICAL_PILOT_TEST_REMOTE_CORES=32 \
RADICAL_PILOT_TEST_REMOTE_NUM_CUS=64 \
python setup.py test
```

Note: Be aware that it can take quite some time for pilots to get scheduled on the remote system. You can set `RADICAL_PILOT_TEST_TIMEOUT` to force the tests to abort after a given number of minutes.

1.10.3 Adding New Tests

If you want to add a new test, for example to reproduce an error that you have encountered, please follow this procedure:

In the `src/radical/pilot/tests/issues/` directory, create a new file. If applicable, name it after the issues number in the RADICAL-Pilot issues tracker, e.g., `issue_123.py`.

The content of the file should look like this (make sure to change the class name):

```
import os
import sys
import radical.pilot
import unittest

# DBURL defines the MongoDB server URL and has the format mongodb://
↪user:password@host:port.
# For the installation of a MongoDB server, refer to the MongoDB website:
# http://docs.mongodb.org/manual/installation/
DBURL = os.getenv("RADICAL_PILOT_DBURL")
if DBURL is None:
    print "ERROR: RADICAL_PILOT_DBURL (MongoDB server URL) is not defined."
    sys.exit(1)

DBNAME = 'radicalpilot_unittests'

#-----------------------------------------------------------------------------
#
class TestIssue123(unittest.TestCase):

    def setUp(self):
        # clean up fragments from previous tests
        client = MongoClient(DBURL)
        client.drop_database(DBNAME)

    def tearDown(self):
        # clean up after ourselves
        client = MongoClient(DBURL)
        client.drop_database(DBNAME)

    #-------------------------------------------------------------------------
    #
    def test__issue_163_part_1(self):
        """ https://github.com/radical-cybertools/radical.pilot/issues/123
        """
        session = radical.pilot.Session(database_url=DBURL, database_name=DBNAME)

        # Your test implementation

        session.close()
```

Now you can re-install RADICAL-Pilot and run you new test. In the source root, run:

```
easy_install . && python -m unittest -v -q radical.pilot.tests.issues.issue_123.
↪TestIssue123
```

1.11 Benchmarks

Performance, and specifically improved application performance, is a main objective for the existence of RADICAL-Pilot. To enable users to understand performance of both RADICAL-Pilot itself and of the applications executed with RADICAL-Pilot, we provide some utilities for benchmarking and performance analysis.

Note: Performance profiling is enabled by setting *RADICAL_PILOT_PROFILE* in the application environment. If profiling is enabled, the application can request any number of cores on the resource *local.localhost*.

During operation, RADICAL-Pilot stores time stamps of different events and activities in MongoDB, under the ID of the *radical.pilot.Session*. That information can be used for post mortem performance analysis. To do so, one needs to specify the session ID to be examined – you can print the session ID when running your application, via

```
print "session id: %s" % session.uid
```

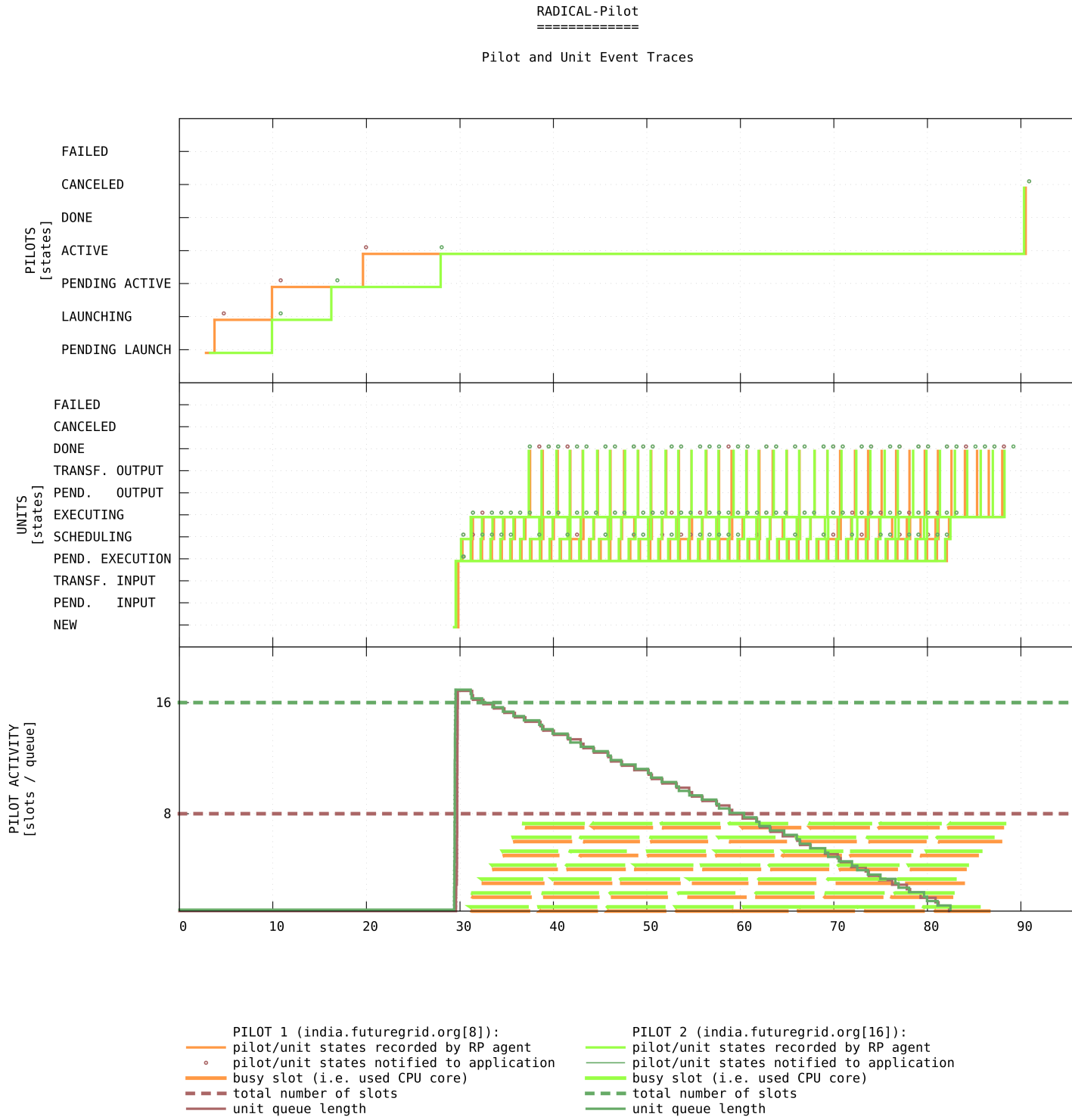
With that session ID, you can use the tool *radicalpilot-stats* to print some statistics, and to plot some performance graphs:

```
$ radicalpilot-stats -m plot -s 53b5bbd174df926f4a4d3318
```

This command will, in the *plot* mode shown above, produce a *53b5bbd174df926f4a4d3318.png* and a *53b5bbd174df926f4a4d3318.pdf* plot (where *53b5bbd174df926f4a4d3318* is the session ID as mentioned. The same command has other modi for inspecting sessions – you can see a help message via

```
$ ./bin/radicalpilot-stats -m help

usage   : ./bin/radicalpilot-stats -m mode [-d dburl] [-s session]
example : ./bin/radicalpilot-stats -m stats -d mongodb://localhost/radicalpilot -s␣
↪536afe101d41c83696ea0135

modes :

  help  : show this message
  list  : show  a  list   of sessions in the database
  tree  : show  a  tree   of session objects
  dump  : show  a  tree   of session objects, with full details
  sort  : show  a  list   of session objects, sorted by type
  hist  : show timeline   of session history
  stat  : show statistics of session history (not implemented)
  plot  : save gnuplot representing session history

The default command is 'list'.  If no session ID is specified, operations
which apply to a single session will choose the last session in the given
DB.  The default MongoDB is 'mongodb://ec2-184-72-89-141.compute-1.amazonaws.
↪com:27017/radicalpilot/'
```

An exemplar performance plot is included below. It represents a number of events and metrics, represented over a time axis. In particular, it shows (at the bottom) the utilization of the various compute cores managed by the pilots in the session – if that utilization is showing no major gaps, your application should make efficient use of the allocated resources.



Note that the plotting capability needs an up-to-date installation of gnuoplot with the cairo-png backend. For Linux, that can be installed from the usual package repositories. For MacOS, the following should take care of the installation:

1.12 Details on Profiling

Note: This section is for developers, and should be disregarded for production runs and 'normal' users in general.

RADICAL-Pilot allows to tweak the pilot process behavior in many details, and specifically allows to artificially increase the load on individual components, for the purpose of more detailed profiling, and identification of bottlenecks. With that background, a pilot description supports an additional attribute *_config*, which accepts a dict of the following structure:

```
pdesc = rp.ComputePilotDescription()
pdesc.resource = "local.localhost"
pdesc.runtime  = 5 # minutes
pdesc.cores    = 8
pdesc.cleanup  = False
pdesc._config  = {'number_of_workers' : {'StageinWorker'   :  1,
                                         'ExecWorker'      :  2,
                                         'StageoutWorker'  :  1,
                                         'UpdateWorker'    :  1},
                  'blowup_factor'     : {'Agent'           :  1,
                                         'stagein_queue'   :  1,
                                         'StageinWorker'   :  1,
                                         'schedule_queue'  :  1,
                                         'Scheduler'       :  1,
                                         'execution_queue' : 10,
                                         'ExecWorker'      :  1,
                                         'watch_queue'     :  1,
                                         'Watcher'         :  1,
                                         'stageout_queue'  :  1,
                                         'StageoutWorker'  :  1,
                                         'update_queue'    :  1,
                                         'UpdateWorker'    :  1},
                  'drop_clones'       : {'Agent'           :  1,
                                         'stagein_queue'   :  1,
                                         'StageinWorker'   :  1,
                                         'schedule_queue'  :  1,
                                         'Scheduler'       :  1,
                                         'execution_queue' :  1,
                                         'ExecWorker'      :  0,
                                         'watch_queue'     :  0,
                                         'Watcher'         :  0,
                                         'stageout_queue'  :  1,
                                         'StageoutWorker'  :  1,
                                         'update_queue'    :  1,
                                         'UpdateWorker'    :  1}}
```

That configuration tunes the concurrency of some of the pilot components (here we use two *ExecWorker* instances to spawn units. Further, we request that the number of compute units handled by the *ExecWorker* is 'blown up' (multiplied) by 10. This will created 9 near-identical units for every unit which enters that component, and thus the load increases on that specific component, but not on any of the previous ones. Finally, we instruct all components but the *ExecWorker*, *watch_queue* and *Watcher* to drop the clones again, so that later components won't see those clones eiter. We thus strain only a specific part of the pilot.

Setting these parameters requires some understanding of the pilot architecture. While in general the application semantics remains unaltered, these parameters do significantly alter resource consumption. Also, there do exist invalid combinations which will cause the agent to fail, specifically it will usually be invalid to push updates of cloned units to the client module (via MongoDB).

The pilot profiling (as stored in *agent.prof* in the pilot sandbox) will contain timings for the cloned units. The unit IDs will be based upon the original unit IDs, but have an appendix *.clone.0001* etc., depending on the value of the respective blowup factor. In general, only one of the blowup-factors should be larger than one (otherwise the number of units will grow exponentially, which is probably not what you want).

1.13 Frequently Asked Questions

Here are some answers to frequently-asked questions. Got a question that isn't answered here? Try the mailing list, or file an issue bug tracker.

1.13.1 How do I...

...avoid the error "OperationFailure: too many namespaces/collections"

```
Traceback (most recent call last):
  File "application.py", line 120, in __init__
    db_connection_info=session._connection_info)
  File "/lib/python2.7/site-packages/radical/pilot/controller/pilot_manager_
↪controller.py", line 88, in __init__
    pilot_launcher_workers=pilot_launcher_workers
  File "/lib/python2.7/site-packages/radical/pilot/db/database.py", line 253, in␣
↪insert_pilot_manager
    result = self._pm.insert(pilot_manager_json)
  File "build/bdist.linux-x86_64/egg/pymongo/collection.py", line 412, in insert
  File "build/bdist.linux-x86_64/egg/pymongo/mongo_client.py", line 1121, in _send_
↪message
  File "build/bdist.linux-x86_64/egg/pymongo/mongo_client.py", line 1063, in __check_
↪response_to_last_error
pymongo.errors.OperationFailure: too many namespaces/collections
```

This can happen if radical.pilot too many sessions are piling up in the back-end database. Normally, all database entries are removed when a RADICAL-Pilot session is closed via `session.close()` (or more verbose via `session.close(cleanup=True)`, which is the default. However, if the application fails and is not able to close the session,

or if the session entry remains puprosefully in place for later analysis with `radicalpilot-stats`, then those entries add up over time.

RADICAL-Pilot provides two utilities which can be used to address this problem: `radicalpilot-close-session` can be used to close a session when it is not used anymore; `radicalpilot-cleanup` can be used to clean up all sessions older than a specified number of hours or days, to purge orphaned session entries in a bulk.

. . . avoid the error "Permission denied (publickey,keyboard-interactive)." in AGENT.STDERR or STDERR.

The AGENT.STDERR file or the STDERR file in the unit directory shows the following error and the pilot or unit never starts running:

```
Permission denied (publickey,keyboard-interactive).
kill: 19932: No such process
```

Even though this should already be set up by default on many HPC clusters, it is not always the case. The following instructions will help you to set up password-less SSH between the cluster nodes correctly.

Log-in to the **head-node** or **login-node** of the cluster and run the following commands:

```
cd ~/.ssh/
ssh-keygen -t rsa
```

Do not enter a passphrase. The result should look something like this:

```
Generating public/private rsa key pair.
Enter file in which to save the key (/home/e290/e290/oweidner/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/e290/e290/oweidner/.ssh/id_rsa.
Your public key has been saved in /home/e290/e290/oweidner/.ssh/id_rsa.pub.
The key fingerprint is:
73:b9:cf:45:3d:b6:a7:22:72:90:28:0a:2f:8a:86:fd oweidner@eslogin001
```

Next, add you newly generated key to ~/.ssh/authorized_keys:

```
cat id_rsa.pub >> ~/.ssh/authorized_keys
```

This should be all. Next time you run radical.pilot, you shouldn't see that error message anymore.

(For more general information on SSH keys, check out this link: http://www.linuxproblem.org/art_9.html)

. . . avoid the error "Failed to execvp() 'mybinary': No such file or directory (2)"

This may happen specifically on Gordon. The full error in STDERR is something like:

```
[gcn-X-X.sdsc.edu:mpispawn_0][spawn_processes] Failed to execvp() 'mybinary': No such␣
↪file or directory (2)
```

You need to specify the full path of the executable as mpirun_rsh is not able to find it in the path

. . . avoid errors from setuptools when trying to use a virtualenv?

This happens most likely because an upgrade of pip or setuptools failed.

We have seen occurences where an update of setuptools or pip can make a virtualenv unusable. We don't have any suggestion on how to get the affected virtualenv clean again - it seems easiest to just start over with a new virtualenv. If the problem persists, try to use the default version of setuptools and pip, i.e. do not upgrade them.

. . . avoid the error "Received message too long 1903391841"

This error may show up in the DEBUG level logs during file staging or pilot startup, when sftp is used as a transfer protocol. We have seen this error being caused by verbose *.bashrc* files (or other login files), which confuses sftp startup. Please make sure that any parts of the bashrc which print information etc. are only executed on interactive shell (ie. on shells which have a prompt set as *$PS1*). The snippet below shows how to di that:

```
if [ ! -z "$PS1" ]
then
  echo "hello $USER"
  date
fi
```

. . . avoid the pop-up "Do you want the application python to accept incoming network connections?" on Mac OSX.

This is coming from the firewall on your Mac. You can either:

- click "Allow" (many times)
- disable your firewall (temporarily)
- Sign the application per instructions here: http://apple.stackexchange.com/a/121010

. . . avoid the error "Could not detect shell prompt (timeout)"

We generally only support *sh* and *bash* as login shells on the target machines. Please try to switch to those shells if you use others like *zsh* and *csh/tcsh*. If you need other shells supported, please open a ticket.

Prompt detecting behaviour can be improved by calling *touch $HOME/.hushlogin* on the target machine, which will suppress some system messages on login.

If the problem persists, please open a ticket.

Details: we implement rather cumbersome screen scraping via an interactive ssh session to get onto the target machine, instead of using *paramiko* or other modules. This gives us better performance, but most importantly, this gives us support for *gsissh*, which we did not find available in any other package so far.

1.13.2 Other Questions

How many concurrent RADICAL-Pilot scripts can I execute?

From a RADICAL-Pilot perspective there is no limit, but as SSH is used to access many systems, there is a resource specific limit of the number of SSH connections one can make.

1.14 Developer Documentation

1.14.1 Installation from Source

If you are planning to contribute to the RADICAL-Pilot codebase, or if you want to use the latest and greatest development features, you can download and install RADICAL-Pilot directly from the sources.

First, you need to check out the sources from GitHub.

```
git clone https://github.com/radical-cybertools/radical.pilot.git
```

Next, run the installer directly from the source directoy (assuming you have set up a vritualenv):

```
pip install --upgrade .
```

1.14.2 License

RADICAL-Pilot uses the MIT License (https://github.com/radical-cybertools/radical.pilot/blob/devel/LICENSE.md).

1.14.3 Style Guide

To maintain consistency and uniformity we request people to try to follow our coding style guide lines.

We generally follow PEP 8 (http://legacy.python.org/dev/peps/pep-0008/), with currently one explicit exception:

- When alignment of assignments improves readability.

1.14.4 Debugging

The *RADICAL_PILOT_VERBOSE* environment variable controls the debug output of a RADICAL-Pilot application. Possible values are:

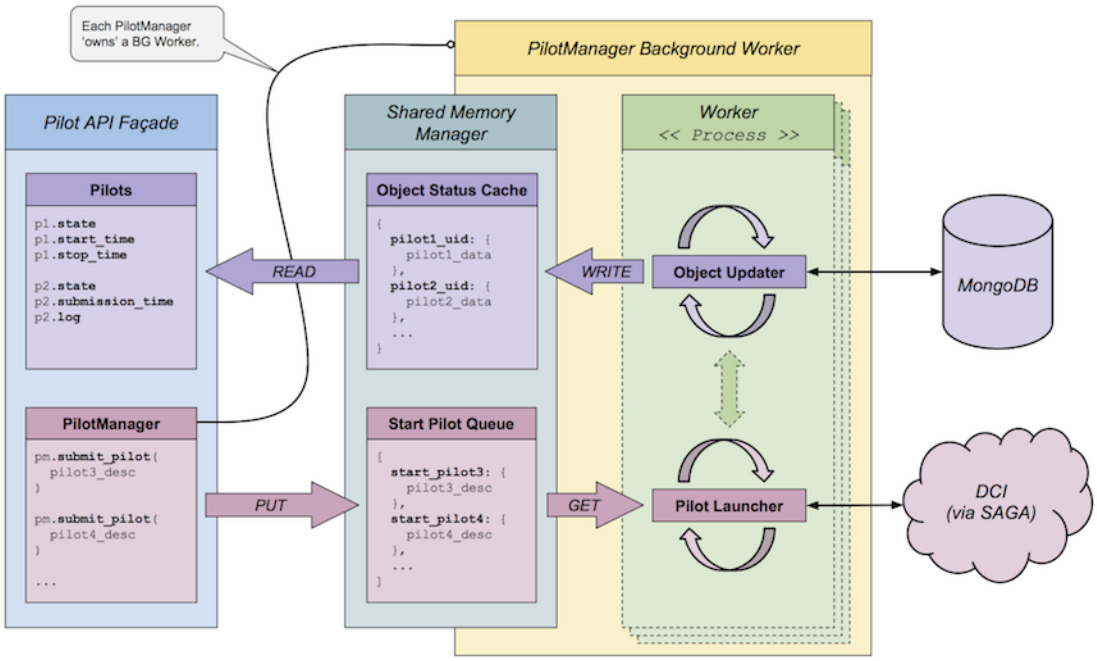
- *debug*
- *info*
- *warning*
- *error*

The environment variable *RADICAL_PILOT_AGENT_VERBOSE* controls the debug log level of the agent process on the target resource. If it is not set, the log level from *RADICAL_PILOT_VERBOSE* is used.

1.14.5 RADICAL-Pilot Architecture

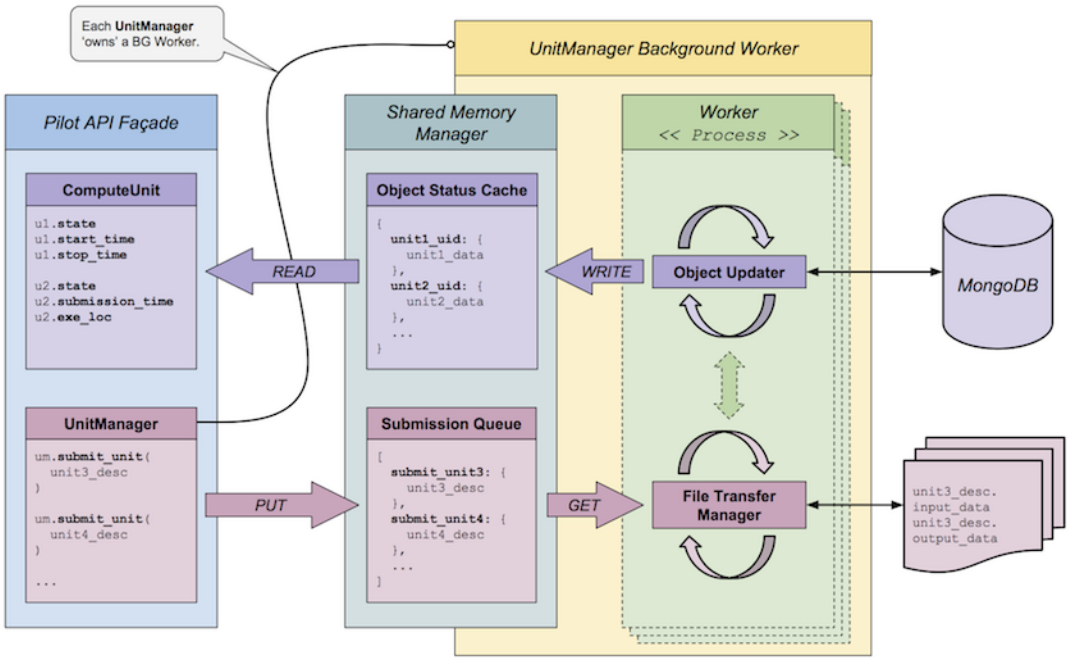
Describe architecture overview here.

PilotManager and PilotManager Worker



Download PDF version.

UnitManager and UnitManager Worker



Download PDF version.

Index

Symbols

A

C

D

E

F

G

I

L

M

N

P

R

S

U

V

W