RADICAL-Pilot Documentation

Release 0.23

The RADICAL Group at Rutgers University

October 18, 2015

Contents

RADICAL-Pilot 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, the resource is acquired by a user's application so that the application can schedule `ComputeUnits` into that resource directly, 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`.

Mailing Lists

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

CHAPTER 1

Contents:

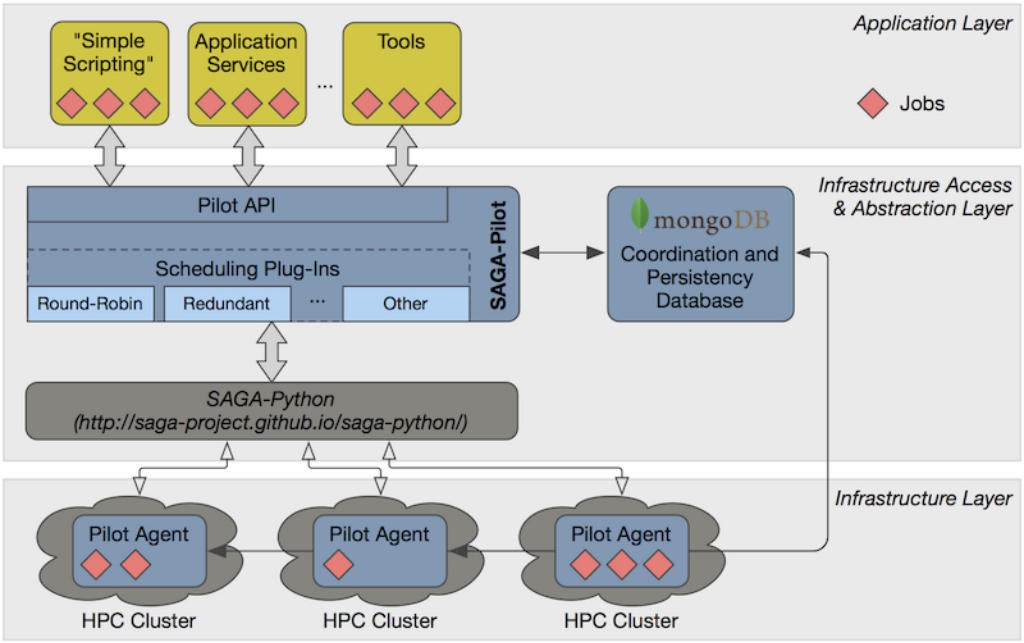
1.1 Introduction

RADICAL-Pilot 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, the resource is acquired by a user's application so that the application can schedule `ComputeUnits` into that resource directly, 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 SAGA- Pilot also supports execution of parallel executables, for example based on MPI or OpenMP.



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.

1.2 Installation

1.2.1 Requirements

RADICAL-Pilot needs Python >= 2.7. 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 can 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.

1.2.2 Installation

To install RADICAL-Pilot in a virtual environment, open a terminal and run:

```
virtualenv $HOME/myenv
source $HOME/myenv/bin/activate
pip install radical.pilot
```

Next, you can do a quick sanity check to make sure that the the packages have been installed properly. In the same virtualenv, run:

```
radicalpilot-version
```

1.2.3 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.g
tar xzf virtualenv-1.9.tar.gz

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

TypeError: 'NoneType' object is not callable

Note that some Python installations have a broken multiprocessing module – if you experience the following error during installation:

```
Traceback (most recent call last):
  File "/usr/lib/python2.7/atexit.py", line 24, in _run_exitfuncs
    func(*targs, **kargs)
  File "/usr/lib/python2.7/multiprocessing/util.py", line 284, in _exit_function
    info('process shutting down')
TypeError: 'NoneType' object is not callable

you may need to move to Python 2.7 (see http://bugs.python.org/issue15881).
```

1.3 Using Local and Remote HPC Resources

1.3.1 Introduction

The real advantage of using RADICAL-Pilot becomes visible when it is used on large HPC clusters. RADICAL-Pilot allows you to launch a ComputePilot allocating a large number of cores and then use it to run many ComputeUnits with small core-counts. This is not only a very nice abstraction to separate resource allocation / management from resource usage, but also circumvents very effectively HPC cluster queue policies and waiting times which can significantly reduce the total time to completion (TTC) of your application.

If you want to use a remote HPC resource, in this example a cluster named "archer", located at EPSRC, UK, you 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 dealing with 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 5) provides some details about SSH set-up. *Pre-Configured Resources* (page 5) list the resource keys that are already defined and ready to use in RADICAL-Pilot.

1.3.2 Configuring SSH Access

If you can manually SSH into the target resource, RADICAL-Pilot can do the same. While RADICAl-Pilot supports username / 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 this, 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 of the system messages you see on your screen at every login. This can help if you encounter random connection problems with RADICAL-Pilot.

1.3.3 Pre-Configured Resources

Resource configurations are a set of dictionaries that hide specific details of a remote resource (queuing-, file-system- and environment-details) from the user. A user allocates a pre-configured resource like this:

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

We maintain a growing set of resource configuration files. Several of the settings included there can be adapted in the *PilotDescription* (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/*.

1.3.4 Writing a Custom Resource Configuration File

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

Note: Be advised that you may need system admin level knowledge for the target cluster to do so. Also, while RADICAL-Pilot can handle very different types of systems and batch system, it may run into trouble on specific configurationsor versions we did not encounter before. If you run into trouble using a cluster not in our list of officially supported ones, please drop us a note on the 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"              : ["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"],
        "pilot_agent"                : "radical-pilot-agent-multicore.py"
    },
    "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 which 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 (*LOADL*, *LSF*, *PBSPRO*, *SGE*, *SLURM*, *TORQUE*, *FORK*)
- *task_launch_method* : type of compute node access (required for non-MPI units: *SSH*,'APRUN' or *LOCAL*)
- *mpi_launch_method* : type of MPI support (required for MPI units: *MPIRUN*, *MPIEXEC*, *APRUN*, *IBRUN* or *POE*)
- *python_interpreter* : path to python (optional)
- *pre_bootstrap* : list of commands to execute for initialization (optional)
- *valid_roots* : list of shared file system roots (optional). Pilot sandboxes must lie under these roots.
- *pilot_agent* : type of pilot agent to use (*radical-pilot-agent-multicore.py*)
- *forward_tunnel_endpoint* : name of 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/*.

1.3.5 Customizing Resource Configurations Programatically

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

```
import radical.pilot as rp

# create a new session, and get the respective resource config instance
session = rp.Session()
cfg = session.get_resource_config('epsrc.archer')
print "Default queue of archer is: %s" % cfg['default_queue']

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

# now add the entry back.  As we did not change the config name, this will
# replace the original configuration.  A completely new configuration would
# need a unique name.
session.add_resource_config(new_cfg)

# verify that the changes are in place: retrieve the config again and print
# the queue
check_cfg = session.get_resource_config('epsrc.archer')
print "Default queue of archer after change is: %s." % s['default_queue']
```

1.4 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.

1.4.1 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 10) 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 or TRANSFER (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), or `TRANSFER` (to a remote resource).
- **`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 12) 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 16) example demonstrates this in full glory.

1.4.2 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 14) example demonstrates this in full glory.

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

1.4.3 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 9) 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/exar
```

```
import os
import sys
import radical.pilot as rp

# READ: The RADICAL-Pilot documentation:
#   http://radicalpilot.readthedocs.org/en/latest
#
# Try running this example with RADICAL_PILOT_VERBOSE=debug set if
# you want to see what happens behind the scenes!
```

```
#------------------------------------------------------------------------------
#
def pilot_state_cb (pilot, state) :
    """ this callback is invoked on all pilot state changes """

    print "[Callback]: ComputePilot '%s' state: %s." % (pilot.uid, state)

    if  state == rp.FAILED :
        sys.exit (1)


#------------------------------------------------------------------------------
#
def unit_state_cb (unit, state) :
    """ this callback is invoked on all unit state changes """

    print "[Callback]: ComputeUnit '%s' state: %s." % (unit.uid, state)

    if  state == rp.FAILED :
        sys.exit (1)


#------------------------------------------------------------------------------
#
if __name__ == "__main__":

    # Create a new session. A session is the 'root' object for all other
    # RADICAL-Pilot objects. It encapsulates the MongoDB connection(s) as
    # well as security credentials.
    session = rp.Session()

    # Add a Pilot Manager. Pilot managers manage one or more ComputePilots.
    pmgr = rp.PilotManager(session=session)

    # Register our callback with the PilotManager. This callback will get
    # called every time any of the pilots managed by the PilotManager
    # change their state.
    pmgr.register_callback(pilot_state_cb)

    # Define a single-core local pilot that runs for 5 minutes and cleans up
    # after itself.
    pdesc = rp.ComputePilotDescription()
    pdesc.resource = "local.localhost"
    pdesc.cores    = 1
    pdesc.runtime  = 5 # Minutes
    #pdesc.cleanup  = True

    # Launch the pilot.
    pilot = pmgr.submit_pilots(pdesc)

    # Create a Compute Unit that sorts the local password file and writes the
    # output to result.dat.
    #
    #  The exact command that is executed by the agent is:
    #    "/usr/bin/sort -o result.dat passwd"
    #
    cud = rp.ComputeUnitDescription()
    cud.executable     = "/usr/bin/sort"
```

```
    cud.arguments      = ["-o", "result.dat", "passwd"]
    cud.input_staging  = "/etc/passwd"
    cud.output_staging = "result.dat"

    # Combine the ComputePilot, the ComputeUnits and a scheduler via
    # a UnitManager object.
    umgr = rp.UnitManager(session, rp.SCHED_DIRECT_SUBMISSION)

    # Register our callback with the UnitManager. This callback will get
    # called every time any of the units managed by the UnitManager
    # change their state.
    umgr.register_callback(unit_state_cb)

    # Add the previously created ComputePilot to the UnitManager.
    umgr.add_pilots(pilot)

    # Submit the previously created ComputeUnit description to the
    # PilotManager. This will trigger the selected scheduler to start
    # assigning the ComputeUnit to the ComputePilot.
    unit = umgr.submit_units(cud)

    # Wait for the compute unit to reach a terminal state (DONE or FAILED).
    umgr.wait_units()

    print "* Task %s (executed @ %s) state: %s, exit code: %s, started: %s, " \
          "finished: %s, output file: %s" % \
          (unit.uid, unit.execution_locations, unit.state,
           unit.exit_code,  unit.start_time, unit.stop_time,
           unit.description.output_staging[0]['target'])

    # Close automatically cancels the pilot(s).
    session.close()

# ------------------------------------------------------------------------------
```

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/exar
```

```
import os
import sys
import radical.pilot as rp

# READ: The RADICAL-Pilot documentation:
#   http://radicalpilot.readthedocs.org/en/latest
#
# Try running this example with RADICAL_PILOT_VERBOSE=debug set if
# you want to see what happens behind the scenes!
```

```
#------------------------------------------------------------------------------
#
def pilot_state_cb (pilot, state) :
    """ this callback is invoked on all pilot state changes """

    print "[Callback]: ComputePilot '%s' state: %s." % (pilot.uid, state)

    if  state == rp.FAILED :
        sys.exit (1)


#------------------------------------------------------------------------------
#
def unit_state_cb (unit, state) :
    """ this callback is invoked on all unit state changes """

    print "[Callback]: ComputeUnit '%s' state: %s." % (unit.uid, state)

    if  state == rp.FAILED :
        sys.exit (1)


#------------------------------------------------------------------------------
#
if __name__ == "__main__":

    # Create a new session. A session is the 'root' object for all other
    # RADICAL-Pilot objects. It encapsulates the MongoDB connection(s) as
    # well as security credentials.
    session = rp.Session()

    # Add a Pilot Manager. Pilot managers manage one or more ComputePilots.
    pmgr = rp.PilotManager(session=session)

    # Register our callback with the PilotManager. This callback will get
    # called every time any of the pilots managed by the PilotManager
    # change their state.
    pmgr.register_callback(pilot_state_cb)

    # Define a single-core local pilot that runs for 5 minutes and cleans up
    # after itself.
    pdesc = rp.ComputePilotDescription()
    pdesc.resource = "local.localhost"
    pdesc.cores    = 8
    pdesc.runtime  = 5 # Minutes
    #pdesc.cleanup  = True

    # Launch the pilot.
    pilot = pmgr.submit_pilots(pdesc)

    input_sd = {
        'source': '/etc/passwd',
        'target': 'input.dat'
    }

    output_sd = {
        'source': 'result.dat',
        'target': '/tmp/result.dat'
```

```
    }

    # Create a Compute Unit that sorts the local password file and writes the
    # output to result.dat.
    #
    #  The exact command that is executed by the agent is:
    #    "/usr/bin/sort -o result.dat input.dat"
    #
    cud = rp.ComputeUnitDescription()
    cud.executable     = "sort"
    cud.arguments      = ["-o", "result.dat", "input.dat"]
    cud.input_staging  = input_sd
    cud.output_staging = output_sd

    # Combine the ComputePilot, the ComputeUnits and a scheduler via
    # a UnitManager object.
    umgr = rp.UnitManager(session, rp.SCHED_DIRECT_SUBMISSION)

    # Register our callback with the UnitManager. This callback will get
    # called every time any of the units managed by the UnitManager
    # change their state.
    umgr.register_callback(unit_state_cb)

    # Add the previously created ComputePilot to the UnitManager.
    umgr.add_pilots(pilot)

    # Submit the previously created ComputeUnit description to the
    # PilotManager. This will trigger the selected scheduler to start
    # assigning the ComputeUnit to the ComputePilot.
    unit = umgr.submit_units(cud)

    # Wait for the compute unit to reach a terminal state (DONE or FAILED).
    umgr.wait_units()

    print "* Task %s (executed @ %s) state: %s, exit code: %s, started: %s, " \
          "finished: %s, output file: %s" % \
          (unit.uid, unit.execution_locations, unit.state,
           unit.exit_code,  unit.start_time, unit.stop_time,
           unit.description.output_staging[0]['target'])

    # Close automatically cancels the pilot(s).
    session.close()

# ------------------------------------------------------------------------------
```

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/exar
```

```
import os
import radical.pilot
```

```
SHARED_INPUT_FILE = 'shared_input_file.txt'
MY_STAGING_AREA = 'staging_area'

#------------------------------------------------------------------------------
#
if __name__ == "__main__":

    try:

        # Create shared input file
        os.system('/bin/echo -n "Hello world, " > %s' % SHARED_INPUT_FILE)
        radical_cockpit_occupants = ['Alice', 'Bob', 'Carol', 'Eve']

        # Create per unit input files
        for idx, occ in enumerate(radical_cockpit_occupants):
            input_file = 'input_file-%d.txt' % (idx+1)
            os.system('/bin/echo "%s" > %s' % (occ, input_file))

        # Create a new session. A session is the 'root' object for all other
        # RADICAL-Pilot objects. It encapsulates the MongoDB connection(s) as
        # well as security credentials.
        session = radical.pilot.Session()

        # Add a Pilot Manager. Pilot managers manage one or more ComputePilots.
        pmgr = radical.pilot.PilotManager(session=session)

        # Define a C-core on $RESOURCE that runs for M minutes and
        # uses $HOME/radical.pilot.sandbox as sandbox directory.
        pdesc = radical.pilot.ComputePilotDescription()
        pdesc.resource = "xsede.trestles"
        pdesc.runtime  = 5 # M minutes
        pdesc.cores    = 8 # C cores

        # Launch the pilot.
        pilot = pmgr.submit_pilots(pdesc)

        # Define the url of the local file in the local directory
        shared_input_file_url = 'file://%s/%s' % (os.getcwd(), SHARED_INPUT_FILE)

        staged_file = "~/%s/%s" % (MY_STAGING_AREA, SHARED_INPUT_FILE)
        print "############################"
        print staged_file
        print "############################"

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

        # Configure the staging directive for shared input file.
        sd_shared = {'source': staged_file,
                     'target': SHARED_INPUT_FILE,
                     'action': radical.pilot.LINK
        }
```

```
        # Combine the ComputePilot, the ComputeUnits and a scheduler via
        # a UnitManager object.
        umgr = radical.pilot.UnitManager(session, radical.pilot.SCHED_BACKFILLING)

        # Add the previously created ComputePilot to the UnitManager.
        umgr.add_pilots(pilot)

        compute_unit_descs = []

        for unit_idx in range(len(radical_cockpit_occupants)):

            # Configure the per unit input file.
            input_file = 'input_file-%d.txt' % (unit_idx+1)

            # Configure the for per unit output file.
            output_file = 'output_file-%d.txt' % (unit_idx+1)

            # Actual task description.
            # Concatenate the shared input and the task specific input.
            cud = radical.pilot.ComputeUnitDescription()
            cud.executable = '/bin/bash'
            cud.arguments = ['-c', 'cat %s %s > %s' %
                             (SHARED_INPUT_FILE, input_file, output_file)]
            cud.cores = 1
            cud.input_staging = [sd_shared, input_file]
            cud.output_staging = output_file

            compute_unit_descs.append(cud)

        # Submit the previously created ComputeUnit descriptions to the
        # PilotManager. This will trigger the selected scheduler to start
        # assigning ComputeUnits to the ComputePilots.
        units = umgr.submit_units(compute_unit_descs)

        # Wait for all compute units to finish.
        umgr.wait_units()

        for unit in umgr.get_units():

            # Get the stdout and stderr streams of the ComputeUnit.
            print " STDOUT: %s" % unit.stdout
            print " STDERR: %s" % unit.stderr

        session.close()

    except radical.pilot.PilotException, ex:
        print "Error: %s" % ex
```

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/exar
```

```
import os
import radical.pilot

INPUT_FILE = 'input_file.txt'
INTERMEDIATE_FILE = 'intermediate_file.txt'
OUTPUT_FILE = 'output_file.txt'

#------------------------------------------------------------------------------
#
if __name__ == "__main__":

    try:
        # Create input file
        radical_cockpit_occupants = ['Carol', 'Eve', 'Alice', 'Bob']
        for occ in radical_cockpit_occupants:
            os.system('/bin/echo "%s" >> %s' % (occ, INPUT_FILE))

        # Create a new session. A session is the 'root' object for all other
        # RADICAL-Pilot objects. It encapsulates the MongoDB connection(s) as
        # well as security credentials.
        session = radical.pilot.Session()

        # Add a Pilot Manager. Pilot managers manage one or more ComputePilots.
        pmgr = radical.pilot.PilotManager(session)

        # Define a C-core on stamped that runs for M minutes and
        # uses $HOME/radical.pilot.sandbox as sandbox directory.
        pdesc = radical.pilot.ComputePilotDescription()
        pdesc.resource = "local.localhost"
        pdesc.runtime = 15 # M minutes
        pdesc.cores = 2 # C cores

        # Launch the pilot.
        pilot = pmgr.submit_pilots(pdesc)

        # Combine the ComputePilot, the ComputeUnits and a scheduler via
        # a UnitManager object.
        umgr = radical.pilot.UnitManager(
            session=session,
            scheduler=radical.pilot.SCHED_DIRECT_SUBMISSION)

        # Add the previously created ComputePilot to the UnitManager.
        umgr.add_pilots(pilot)

        # Configure the staging directive for intermediate data
        sd_inter_out = {
            'source': INTERMEDIATE_FILE,
            # Note the triple slash, because of URL peculiarities
            'target': 'staging:///%s' % INTERMEDIATE_FILE,
            'action': radical.pilot.COPY
        }

        # Task 1: Sort the input file and output to intermediate file
        cud1 = radical.pilot.ComputeUnitDescription()
        cud1.executable = 'sort'
        cud1.arguments = ['-o', INTERMEDIATE_FILE, INPUT_FILE]
        cud1.input_staging = INPUT_FILE
        cud1.output_staging = sd_inter_out
```

```
        # Submit the first task for execution.
        umgr.submit_units(cud1)

        # Wait for the compute unit to finish.
        umgr.wait_units()

        # Configure the staging directive for input intermediate data
        sd_inter_in = {
            # Note the triple slash, because of URL peculiarities
            'source': 'staging:///%s' % INTERMEDIATE_FILE,
            'target': INTERMEDIATE_FILE,
            'action': radical.pilot.LINK
        }

        # Task 2: Take the first line of the sort intermediate file and write to output
        cud2 = radical.pilot.ComputeUnitDescription()
        cud2.executable = '/bin/bash'
        cud2.arguments = ['-c', 'head -n1 %s > %s' %
                          (INTERMEDIATE_FILE, OUTPUT_FILE)]
        cud2.input_staging = sd_inter_in
        cud2.output_staging = OUTPUT_FILE

        # Submit the second CU for execution.
        umgr.submit_units(cud2)

        # Wait for the compute unit to finish.
        umgr.wait_units()

        session.close()

    except radical.pilot.PilotException, ex:
        print "Error: %s" % ex
```

1.5 Examples & Best Practice

1.5.1 Getting Started

This is where you should start if you are new to RADICAL-Pilot. It is highly recommended that you carefully read and understand all of this before you go off and start developing your own applications.

In this chapter we explain the main components of RADICAL-Pilot and the foundations of their function and their interplay. For your convenience, you can find a fully working example at the end of this page.

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. Throughout this chapter you will also find links to more advanced topics like launching ComputePilots on remote HPC clusters and scheduling.

Note: This chapter assumes that you have successfully installed RADICAL-Pilot on (see chapter *Installation* (page 4)).

Loading the Module

In order to use RADICAL-Pilot in your Python application, you need to import the `radical.pilot` module.

```
import radical.pilot
```

You can check / print the version of your RADICAL-Pilot installation via the `version` property.

```
print radical.pilot.version
```

Creating a Session

A *radical.pilot.Session* (page 51) is the root object for all other objects in RADICAL- Pilot. You can think of it as a *tree* or a *directory structure* with a Session as root. Each Session can have zero or more *radical.pilot.Context* (page 53), *radical.pilot.PilotManager* (page 53) and *radical.pilot.UnitManager* (page 59) attached to it.

```
(~~~~~~~~~~)
(          ) <---- [Session]
( MongoDB )        |
(          )       |---- Context
(__________)       |---- ....
                   |
                   |---- [PilotManager]
                   |      |
                   |      |---- ComputePilot
                   |      |---- ComputePilot
                   |
                   |---- [UnitManager]
                   |      |
                   |      |---- ComputeUnit
                   |      |---- ComputeUnit
                   |      |....
                   |
                   |---- [UnitManager]
                   |      |
                   |      |....
                   |
                   |....
```

A Session also encapsulates the connection(s) to a back end MongoDB server which is the *brain* and *central nervous system* of RADICAL-Pilot. More information about how RADICAL-Pilot uses MongoDB can be found in the *Introduction* (page 3) section.

To create a new Session, the only thing you need to provide is the URL of a MongoDB server:

```
session = radical.pilot.Session(database_url="mongodb://my-mongodb-server.edu:27017")
```

Each Session has a unique identifier (*uid*) and methods to traverse its members. The Session *uid* can be used to disconnect and reconnect to a Session as required. This is covered in *Disconnecting and Reconnecting* (page 29).

```
print "UID           : %s" % session.uid
print "Contexts      : %s" % session.list_contexts()
print "UnitManagers  : %s" % session.list_unit_managers()
print "PilotManagers : %s" % session.list_pilot_managers()
```

Warning: Always call *radical.pilot.Session.close()* (page 51) before your application terminates. This will ensure that RADICAL-Pilot shuts down properly.

Creating a ComputePilot

A *radical.pilot.ComputePilot* (page 57) is responsible for ComputeUnit (task) execution. ComputePilots can be launched either locally or remotely, on a single machine or on one or more HPC clusters. In this example we just use local ComputePilots, but more on remote ComputePilots and how to launch them on HPC clusters can be found in *Launching Remote / HPC ComputePilots* (page 29).

As shown in the hierarchy above, ComputePilots are grouped in *radical.pilot.PilotManager* (page 53) *containers*, so before you can launch a ComputePilot, you need to add a PilotManager to your Session. Just like a Session, a PilotManager has a unique id (*uid*) as well as a traversal method (*list_pilots*).

```
pmgr = radical.pilot.PilotManager(session=session)
print "PM UID        : %s" % pmgr.uid
print "Pilots        : %s" % pmgr.list_pilots()
```

In order to create a new ComputePilot, you first need to describe its requirements and properties. This is done with the help of a *radical.pilot.ComputePilotDescription* (page 56) object. The mandatory properties that you need to define are:

- *resource* - The name (hostname) of the target system or `localhost` to launch a local ComputePilot.
- *runtime* - The runtime (in minutes) of the ComputePilot agent.
- *cores* - The number or cores the ComputePilot agent will try to allocate.

You can define and submit a 2-core local pilot that runs for 5 minutes like this:

```
pdesc = radical.pilot.ComputePilotDescription()
pdesc.resource  = "local.localhost"
pdesc.runtime   = 5 # minutes
pdesc.cores     = 2
```

A ComputePilot is launched by passing the ComputePilotDescription to the `submit_pilots()` method of the PilotManager. This automatically adds the ComputePilot to the PilotManager. Like any other object in RADICAL-Pilot, a ComputePilot also has a unique identifier (`uid`)

```
pilot = pmgr.submit_pilots(pdesc)
print "Pilot UID     : %s" % pilot.uid
```

Warning: Note that `submit_pilots()` is a non-blocking call and that the submitted ComputePilot agent **will not terminate** when your Python scripts finishes. ComputePilot agents terminate only after they have reached their `runtime` limit or if you call *radical.pilot.PilotManager.cancel_pilots()* (page 56) or *radical.pilot.ComputePilot.cancel()* (page 59).

Note: You can change to the ComputePilot sandbox directory (`/tmp/radical.pilot.sandbox` in the above example) to see the raw logs and output files of the ComputePilot agent(s) `[pilot-<uid>]` as well as the working directories and output of the individual ComputeUnits (`[task-<uid>]`).

```
[/<sandbox-dir>/]
|
|----[pilot-<uid>/]
|     |
|     |---- STDERR
|     |---- STDOUT
|     |---- AGENT.LOG
|     |---- [task-<uid>/]
|     |---- [task-<uid>/]
|     |....
```

```
|
|....
```

Knowing where to find these files might come in handy for debugging purposes but it is not required for regular RADICAL-Pilot usage.

Creating ComputeUnits (Tasks)

After you have launched a ComputePilot, you can now generate a few *radical.pilot.ComputeUnit* (page 64) objects for the ComputePilot 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 ComputePilots, a ComputeUnit is described via a *radical.pilot.ComputeUnitDescription* (page 63) object. The mandatory properties that you need to define are:

- `executable` - The executable to launch.
- `arguments` - The arguments to pass to the executable.
- `cores` - The number of cores required by the executable.

For example, you can create a workload of 8 '/bin/sleep' ComputeUnits like this:

```
compute_units = []

for unit_count in range(0, 8):
    cu = radical.pilot.ComputeUnitDescription()
    cu.environment = {"SLEEP_TIME" : "10"}
    cu.executable  = "/bin/sleep"
    cu.arguments   = ["$SLEEP_TIME"]
    cu.cores       = 1

    compute_units.append(cu)
```

Note: The example above uses a single executable that requires only one core. It is however possible to run multiple commands in one ComputeUnit. This is described in *Executing Multiple Commands in a Single ComputeUnit* (page 33). If you want to run multi-core executables, like for example MPI programs, check out *Executing Multicore / Multithreaded ComputeUnits* (page 34).

Input- / Output-File Transfer

Often, a computational task doesn't just consist of an executable with some arguments but also needs some input data. For this reason, a *radical.pilot.ComputeUnitDescription* (page 63) allows the definition of `input_staging` and `output_staging`:

- `input_staging` defines a list of local files that need to be transferred to the execution resource before a ComputeUnit can start running.
- `output_staging` defines a list of remote files that need to be transferred back to the local machine after a ComputeUnit has finished execution.

See *Data Staging* (page 8) for more information on data staging.

Furthermore, a ComputeUnit provides two properties *radical.pilot.ComputeUnit.stdout* (page 65) and *radical.pilot.ComputeUnit.stderr* (page 65) that can be used to access a ComputeUnit's STDOUT and STDERR files after it has finished execution.

Example:

```
cu = radical.pilot.ComputeUnitDescription()
cu.executable    = "/bin/cat"
cu.arguments     = ["file1.dat", "file2.dat"]
cu.cores         = 1
cu.input_staging = ["./file1.dat", "./file2.dat"]
```

Adding Callbacks

Events in RADICAL-Pilot are mostly asynchronous as they happen at one or more distributed components, namely the ComputePilot agents. At any time during the execution of a workload, ComputePilots and ComputeUnits can begin or finish execution or fail with an error.

RADICAL-Pilot provides callbacks as a method to react to these events asynchronously when they occur. ComputePilots, PilotManagers, ComputeUnits and UnitManagers all have a `register_callbacks` method:

- *radical.pilot.UnitManager.register_callback()* (page 62)
- *radical.pilot.PilotManager.register_callback()* (page 56)
- *radical.pilot.ComputePilot.register_callback()* (page 58)
- *radical.pilot.ComputeUnit.register_callback()* (page 65)

A simple callback that prints the state of all pilots would look something like this:

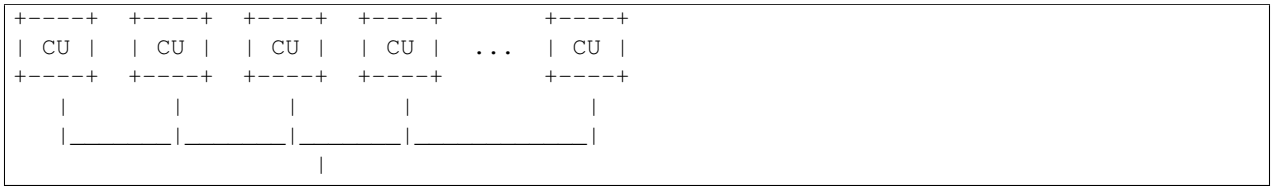
```
def pilot_state_cb(pilot, state):
    print "[Callback]: ComputePilot '%s' state changed to '%s'."% (pilot.uid, state)

pmgr = radical.pilot.PilotManager(session=session)
pmgr.register_callback(pilot_state_cb)
```

Note: Using callbacks can greatly improve the performance of an application since it eradicates the necessity for global / blocking `wait()` calls and state polling. More about callbacks can be read in chapter_programming_with_callbacks.

Scheduling ComputeUnits

In the previous steps we have created and launched a ComputePilot (via a PilotManager) and created a list of ComputeUnitDescriptions. In order to put it all together and execute the ComputeUnits on the ComputePilot, we need to create a *radical.pilot.UnitManager* (page 59) instance.

As shown in the diagram below, a UnitManager combines three things: the ComputeUnits, added via *radical.pilot.UnitManager.submit_units()* (page 61), one or more ComputePilots, added via *radical.pilot.UnitManager.add_pilots()* (page 60) and a *Unit Scheduler* (page 44). Once instantiated, a UnitManager assigns the submitted CUs to one of its ComputePilots based on the selected scheduling algorithm.



```
                    v submit_units()
             +---------------+
             |  UnitManager  |
             |---------------|
             |               |
             |  <SCHEDULER>  |
             +---------------+
                     ^ add_pilots()
                     |
            _________|___________
           |        |            |
        +~~~~+   +~~~~+       +~~~~+
        | CP |   | CP |  ...  | CP |
        +~~~~+   +~~~~+       +~~~~+
```

Since we have only one ComputePilot, we don't need any specific scheduling algorithm for our example. We choose `SCHED_DIRECT_SUBMISSION` which simply passes the ComputeUnits on to the ComputePilot.

```
umgr = radical.pilot.UnitManager(session=session, scheduler=radical.pilot.SCHED_DIRECT_SUBMISSION)

umgr.add_pilots(pilot)
umgr.submit_units(compute_units)

umgr.wait_units()
```

The *radical.pilot.UnitManager.wait_units()* (page 62) call blocks until all ComputeUnits have been executed by the UnitManager. Simple control flows / dependencies can be realized with `wait_units()`, however, for more complex control flows it can become inefficient due to its blocking nature. To address this, RADICAL-Pilot also provides mechanisms for asynchronous notifications and callbacks. This is discussed in more detail in *Application Control Flow with Callbacks* (page 37).

Note: The `SCHED_DIRECT_SUBMISSION` only works with a sinlge ComputePilot. If you add more than one ComputePilot to a UnitManager, you will end up with an error. If you want to use RADICAL-Pilot to run multiple ComputePilots concurrently, possibly on different machines, check out *Launching Remote / HPC ComputePilots* (page 29).

Results and Inspection

```
for unit in umgr.get_units():
    print "unit id  : %s" % unit.uid
    print "  state  : %s" % unit.state
    print "  history:"
    for entry in unit.state_history :
        print "           %s : %s" (entry.timestamp, entry.state)
```

Cleanup and Shutdown

When your application has finished executing all ComputeUnits, it should make an attempt to cancel the ComputePilot. If a ComputePilot is not canceled, it will continue running until it reaches its `runtime` limit, even if application has terminated.

An individual ComputePilot is canceled by calling *radical.pilot.ComputePilot.cancel()* (page 59). Alternatively, all ComputePilots of a PilotManager can be canceled by calling *radical.pilot.PilotManager.cancel_pilots()* (page 56).

```
pmgr.cancel_pilots()
```

Before your application terminates, you should always call *radical.pilot.Session.close()* (page 51) to ensure that your RADICAL-Pilot session terminates properly. If you haven't canceled the pilots before explicitly, `close()` will take care of that implicitly (control it via the *terminate* parameter). `close()` will also delete all traces of the session from the database (control this with the *cleanup* parameter).

```
session.close(cleanup=True, terminate=True)
```

What's Next?

Now that you understand the basic mechanics of RADICAL-Pilot, it's time to dive into some of the more advanced topics. We suggest that you check out the following chapters next:

- *Error Handling Strategies* (page 27). Error handling is crucial for any RADICAL-Pilot application! This chapter captures everything from exception handling to state callbacks.
- *Launching Remote / HPC ComputePilots* (page 29). In this chapter we explain how to launch ComputePilots on remote HPC clusters, something you most definitely want to do.
- *Disconnecting and Reconnecting* (page 29). This chapter is very useful for example if you work with long-running tasks that don't need continuous supervision.

The Complete Example

Below is a complete and working example that puts together everything we discussed in this section. You can download the sources from `here`.

```
import os
import sys
import time
import radical.pilot as rp

# READ: The RADICAL-Pilot documentation:
#   http://radicalpilot.readthedocs.org/en/latest
#
# Try running this example with RADICAL_PILOT_VERBOSE=debug set if
# you want to see what happens behind the scences!


#------------------------------------------------------------------------------
#
def pilot_state_cb (pilot, state) :
    """ this callback is invoked on all pilot state changes """

    print "[Callback]: ComputePilot '%s' state: %s." % (pilot.uid, state)

    if  state == rp.FAILED :
        sys.exit (1)


#------------------------------------------------------------------------------
#
def unit_state_cb (unit, state) :
    """ this callback is invoked on all unit state changes """
```

```
    print "[Callback]: ComputeUnit  '%s: %s' (on %s) state: %s." \
        % (unit.name, unit.uid, unit.pilot_id, state)

    if  state == rp.FAILED :
        print "stderr: %s" % unit.stderr
        sys.exit (1)


#------------------------------------------------------------------------------
#
def wait_queue_size_cb(umgr, wait_queue_size):
    """
    this callback is called when the size of the unit managers wait_queue
    changes.
    """
    print "[Callback]: UnitManager  '%s' wait_queue_size changed to %s." \
        % (umgr.uid, wait_queue_size)

    pilots = umgr.get_pilots ()
    for pilot in pilots :
        print "pilot %s: %s" % (pilot.uid, pilot.state)

    if  wait_queue_size == 0 :
        for pilot in pilots :
            if  pilot.state in [rp.PENDING_LAUNCH,
                                rp.LAUNCHING     ,
                                rp.PENDING_ACTIVE] :
                print "cancel pilot %s" % pilot.uid
                umgr.remove_pilot (pilot.uid)
                pilot.cancel ()


#------------------------------------------------------------------------------
#
if __name__ == "__main__":

    # prepare some input files for the compute units
    os.system ('hostname > file1.dat')
    os.system ('date     > file2.dat')

    # Create a new session. A session is the 'root' object for all other
    # RADICAL-Pilot objects. It encapsulates the MongoDB connection(s) as
    # well as security credentials.
    session = rp.Session()
    print "session id: %s" % session.uid

    # Add a Pilot Manager. Pilot managers manage one or more ComputePilots.
    pmgr = rp.PilotManager(session=session)

    # Register our callback with the PilotManager. This callback will get
    # called every time any of the pilots managed by the PilotManager
    # change their state.
    pmgr.register_callback(pilot_state_cb)

    # Define a 4-core local pilot that runs for 10 minutes and cleans up
    # after itself.
    pdesc = rp.ComputePilotDescription()
    pdesc.resource = "local.localhost"
```

```
    pdesc.runtime  = 5 # minutes
    pdesc.cores    = 1
    pdesc.cleanup  = True

    # Launch the pilot.
    pilot = pmgr.submit_pilots(pdesc)

    # Combine the ComputePilot, the ComputeUnits and a scheduler via
    # a UnitManager object.
    umgr = rp.UnitManager(
        session=session,
        scheduler=rp.SCHED_BACKFILLING)

    # Register our callback with the UnitManager. This callback will get
    # called every time any of the units managed by the UnitManager
    # change their state.
    umgr.register_callback(unit_state_cb, rp.UNIT_STATE)

    # Register also a callback which tells us when all units have been
    # assigned to pilots
    umgr.register_callback(wait_queue_size_cb, rp.WAIT_QUEUE_SIZE)


    # Add the previously created ComputePilot to the UnitManager.
    umgr.add_pilots(pilot)

    # Create a workload of ComputeUnits (tasks). Each compute unit
    # uses /bin/cat to concatenate two input files, file1.dat and
    # file2.dat. The output is written to STDOUT. cu.environment is
    # used to demonstrate how to set environment variables within a
    # ComputeUnit - it's not strictly necessary for this example. As
    # a shell script, the ComputeUnits would look something like this:
    #
    #    export INPUT1=file1.dat
    #    export INPUT2=file2.dat
    #    /bin/cat $INPUT1 $INPUT2
    #
    cuds = []
    for unit_count in range(0, 20):
        cud = rp.ComputeUnitDescription()
        cud.name          = "unit_%03d" % unit_count
        cud.executable    = "/bin/sh"
        cud.environment   = {'INPUT1': 'file1.dat', 'INPUT2': 'file2.dat'}
        cud.arguments     = ["-l", "-c", "cat $INPUT1 $INPUT2"]
        cud.cores         = 1
        cud.input_staging = ['file1.dat', 'file2.dat']

        cuds.append(cud)

    # Submit the previously created ComputeUnit descriptions to the
    # PilotManager. This will trigger the selected scheduler to start
    # assigning ComputeUnits to the ComputePilots.
    units = umgr.submit_units(cuds)

    # Wait for all compute units to reach a terminal state (DONE or FAILED).
    umgr.wait_units()

    print 'units all done'
```

```
    print '------------------------------------------------------------------'

    for unit in units :
        unit.wait ()

    for unit in units:
        print "* Task %s (executed @ %s) state %s, exit code: %s, started: %s, finished: %s, stdout:
            % (unit.uid, unit.execution_locations, unit.state, unit.exit_code, unit.start_time, unit

    # Close automatically cancels the pilot(s).
    session.close (terminate=True)

    # delete the test data files
    os.system ('rm file1.dat')
    os.system ('rm file2.dat')
```

1.5.2 Error Handling Strategies

Error Handling in RADICAL-Pilot

```
#!/usr/bin/env python

__copyright__ = "Copyright 2013-2014, http://radical.rutgers.edu"
__license__   = "MIT"

import os
import sys
import radical.pilot as rp
import time

# READ: The RADICAL-Pilot documentation:
#   http://radicalpilot.readthedocs.org/en/latest
#
# Try running this example with RADICAL_PILOT_VERBOSE=debug set if
# you want to see what happens behind the scences!


#------------------------------------------------------------------------------
#
def pilot_state_cb (pilot, state) :
    """ this callback is invoked on all pilot state changes """

    # Callbacks happen in a different thread than the main application thread --
    # they are truly asynchronous.  That means, however, that a 'sys.exit()'
    # will not end the application, but will end the thread (in this case the
    # pilot_manager_controller thread).  For that reason we wrapped all threads
    # in their own try/except clauses, and then translate the `sys.exit()` into an
    # 'thread.interrupt_main()' call -- this will raise a 'KeyboardInterrupt' in
    # the main thread which can be interpreted by your application, for example
    # to initiate a clean shutdown via `session.close()` (see code later below.)
    # The same `KeyboardShutdown` will also be raised when you interrupt the
    # application via `^C`.
    #
    # Note that other error handling semantics is available, depending on your
    # application's needs.  The application could for example spawn
    # a replacement pilot at this point, or reduce the number of compute units
```

```
    # to match the remaining set of pilots.

    print "[Callback]: ComputePilot '%s' state: %s." % (pilot.uid, state)

    if  state == rp.FAILED :
        print 'Pilot failed -- ABORT!  ABORT!  ABORT!'
        print pilot.log[-1] # Get the last log message
        sys.exit (1)


#-------------------------------------------------------------------------------
#
def unit_state_cb (unit, state) :
    """ this callback is invoked on all unit state changes """

    # The principle for unit state callbacks is exactly the same as for the
    # pilot state callbacks -- only that they are invoked by the unit manager on
    # changes of compute unit states.
    #
    # The example below does not really create any ComputeUnits, we only include
    # the callback here for documentation on the principles of error handling.
    #
    # Note that other error handling semantics is available, depending on your
    # application's needs.  The application could for example spawn replacement
    # compute units, or spawn a pilot on a different resource which might be
    # better equipped to handle the unit payload.

    print "[Callback]: ComputeUnit '%s' state: %s." % (unit.uid, state)

    if  state == rp.FAILED :
        print 'Unit failed -- ABORT!  ABORT!  ABORT!'
        print unit.stderr # Get the unit's stderr
        sys.exit (1)


#-------------------------------------------------------------------------------
#
if __name__ == "__main__":

    """
    This example shows how simple error handling can be implemented
    synchronously using blocking wait() calls.

    The code launches a pilot with 128 cores on 'localhost'. Unless localhost
    has 128 or more cores available, this is bound to fail. This example shows
    how this error can be caught and handled.
    """

    # Create a new session. No need to try/except this: if session creation
    # fails, there is not much we can do anyways...
    session = rp.Session()


    # all other pilot code is now tried/excepted.  If an exception is caught, we
    # can rely on the session object to exist and be valid, and we can thus tear
    # the whole RP stack down via a 'session.close()' call in the 'finally'
    # clause...
    try :
```

```
        # do pilot thingies
        pmgr = rp.PilotManager(session=session)

        # Register our callback with the PilotManager. This callback will get
        # called every time any of the pilots managed by the PilotManager
        # change their state -- in particular also on failing pilots.
        pmgr.register_callback(pilot_state_cb)

        # Create a local pilot with a million cores. This will most likely
        # fail as not enough cores will be available.  That means the pilot will
        # go quickly into failed state, and trigger the callback from above.
        pd = rp.ComputePilotDescription()
        pd.resource  = "local.localhost"
        pd.cores     = 1000000
        pd.runtime   = 60

        pilot = pmgr.submit_pilots(pd)

        # this will basically wait forever (the pilot won't reach DONE state...
        state = pilot.wait (state=[rp.DONE])

    except Exception as e :
        # Something unexpected happened in the pilot code above
        print "caught Exception: %s" % e

    except (KeyboardInterrupt, SystemExit) as e :
        # the callback called sys.exit(), and we can here catch the
        # corresponding KeyboardInterrupt exception for shutdown.  We also catch
        # SystemExit (which gets raised if the main threads exits for some other
        # reason).
        print "need to exit now: %s" % e

    finally :
        # always clean up the session, no matter if we caught an exception or
        # not.
        print "closing session"
        session.close ()

        # the above is equivalent to
        #
        #   session.close (cleanup=True, terminate=True)
        #
        # it will thus both clean out the session's database record, and kill
        # all remaining pilots (none in our example).


#-------------------------------------------------------------------------------
```

1.5.3 Disconnecting and Reconnecting

1.5.4 Launching Remote / HPC ComputePilots

This chapter describes how to use RADICAL-Pilot to execute ComputeUnits on ComputePilots running on one or more distributed HPC clusters.

As a pilot-job system, RADICAL-Pilot aims to provide a programmable resource overlay that allows a user to effi-

ciently execute their workloads (tasks) transparently across one or more distributed resources.

RADICAL-Pilot has been developed with HPC (High-Performance Computing) clusters as the primary type of distributed resources in mind. Currently RADICAL-Pilot supports HPC clusters running the following queuing systems:

- PBS / PBS Pro
- LSF
- SLURM
- Sun Grid Engine
- IBM LoadLeveler

Note: RADICAL-Pilot also provides limited support for Grid-style resources based on HTCondor. For more information checkout chapter_example_condor_grids.

Authentication and Security Contexts

RADICAL-Pilot's remote capabilities are built to a large extend on top of SSH and SFTP. ComputePilot agents are transferred on-the-fly via SFTP and launched via SSH on the remote clusters. Once a ComputePilot agent has been started, the rest of the communication between RADICAL-Pilot and the agent happens through MongoDB (see diagram below).

In order to allow RADICAL-Pilot to launch ComputePilot agents on a remote host via SSH, you need to provided it with the right credentials. This is done via the *radical.pilot.Context* (page 53) class.

Note: In order for Context to work, you need to be able to manually SSH into the target host, i.e., you need to have either a username and password or a public / private key set for the host. The most practical way is to set up password-less public-key authentication on the remote host. More about password-less keys can be found HERE.

Assuming that you have password-less public-key authentication set up for a remote host, the most common way to use Context is to set the user name you use on the remote host:

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

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

Once you have added a credential to a session, it is available to all PilotManagers that are created withing this session.

Launching an HPC ComputePilot

You can describe a *radical.pilot.ComputePilot* (page 57) via a *radical.pilot.ComputePilotDescription* (page 56) to the PilotManager:

```
pdesc = radical.pilot.ComputePilotDescription()
pdesc.resource  = "xsede.stampede"
pdesc.runtime   = 15
pdesc.cores     = 32

pilot = pmgr.submit_pilots(pdesc)
```

Launching Multiple ComputePilots

Scheduling ComputeUnits Across Multiple ComputePilots

The Complete Example

Warning: Make sure to adjust ... before you attempt to run it.

```
import os
import sys
import datetime
import radical.pilot as rp

# READ: The RADICAL-Pilot documentation:
#   http://radicalpilot.readthedocs.org/en/latest
#
# Try running this example with RADICAL_PILOT_VERBOSE=debug set if
# you want to see what happens behind the scenes!


#------------------------------------------------------------------------------
#
def pilot_state_cb (pilot, state) :
    """ this callback is invoked on all pilot state changes """

    print "[Callback]: ComputePilot '%s' state changed to %s at %s." % (pilot.uid, state, datetime.da

    if  state == rp.FAILED:
        sys.exit (1)

#------------------------------------------------------------------------------
#
def unit_state_cb (unit, state) :
    """ this callback is invoked on all unit state changes """

    print "[Callback]: ComputeUnit '%s' state changed to %s at %s." % (unit.uid, state, datetime.date

    if state in [rp.FAILED] :
        print "stdout: %s" % unit.stdout
        print "stderr: %s" % unit.stderr


#------------------------------------------------------------------------------
```

```
#
if __name__ == "__main__":

    # prepare some input files for the compute units
    os.system ('head -c 100000 /dev/urandom > file1.dat') # ~ 100k input file
    os.system ('head -c 10000  /dev/urandom > file2.dat') # ~ 10k input file

    # Create a new session. A session is the 'root' object for all other
    # RADICAL-Pilot objects. It encapsulates the MongoDB connection(s) as
    # well as security credentials.
    session = rp.Session()

    # Add an ssh identity to the session.
    c = rp.Context('ssh')
    #c.user_id = "alice"
    #c.user_pass = "ILoveBob!"
    session.add_context(c)

    # Add a Pilot Manager. Pilot managers manage one or more ComputePilots.
    pmgr = rp.PilotManager(session=session)

    # Register our callback with the PilotManager. This callback will get
    # called every time any of the pilots managed by the PilotManager
    # change their state.
    pmgr.register_callback(pilot_state_cb)
    pdescs = list()

    for i in range (1) :
        # Define a 32-core on stampede that runs for 15 minutes and
        # uses $HOME/radical.pilot.sandbox as sandbox directory.
        pdesc = rp.ComputePilotDescription()
        pdesc.resource  = "local.localhost"
        pdesc.runtime   = 15 # minutes
        pdesc.cores     = 8
        pdesc.cleanup   = True
      # pdesc.queue     = "normal"
      # pdesc.project   = "TG-MCB140109"

        pdescs.append (pdesc)

    # Launch the pilot.
    pilots = pmgr.submit_pilots(pdescs)

    # Combine the ComputePilot, the ComputeUnits and a scheduler via
    # a UnitManager object.
    umgr = rp.UnitManager(
        session=session,
        scheduler=rp.SCHED_DIRECT_SUBMISSION)

    # Register our callback with the UnitManager. This callback will get
    # called every time any of the units managed by the UnitManager
    # change their state.
    umgr.register_callback(unit_state_cb)

    # Add the previsouly created ComputePilot to the UnitManager.
    umgr.add_pilots(pilots)

    # Create a workload of 8 ComputeUnits (tasks). Each compute unit
```

```
    # uses /bin/cat to concatenate two input files, file1.dat and
    # file2.dat. The output is written to STDOUT. cu.environment is
    # used to demonstrate how to set environment variables within a
    # ComputeUnit - it's not strictly necessary for this example. As
    # a shell script, the ComputeUnits would look something like this:
    #
    #    export INPUT1=file1.dat
    #    export INPUT2=file2.dat
    #    /bin/cat $INPUT1 $INPUT2
    #
    cuds = list()

    for unit_count in range(0, 32):
        cud = rp.ComputeUnitDescription()
        cud.executable    = "/bin/bash"
        cud.environment   = {'INPUT1': 'file1.dat', 'INPUT2': 'file2.dat'}
        cud.arguments     = ["-l", "-c", "cat $INPUT1 $INPUT2"]
        cud.cores         = 1
        cud.input_staging = ['file1.dat', 'file2.dat']
        cuds.append(cud)

    # Submit the previously created ComputeUnit descriptions to the
    # PilotManager. This will trigger the selected scheduler to start
    # assigning ComputeUnits to the ComputePilots.
    units = umgr.submit_units(cuds)

    # Wait for all compute units to reach a terminal state (DONE or FAILED).
    umgr.wait_units()

    for unit in units:
        print "* Unit %s (executed @ %s) state: %s, exit code: %s, started: %s, finished: %s, output
            % (unit.uid, unit.execution_locations, unit.state, unit.exit_code, unit.start_time, unit
               unit.stdout)

    # Close automatically cancels the pilot(s).
    session.close()

    # delete the test data files
    os.system ('rm file1.dat')
    os.system ('rm file2.dat')

# ------------------------------------------------------------------------------
```

1.5.5 Executing Multiple Commands in a Single ComputeUnit

There are scenarios in which you might want to execute more than one command in a ComputeUnit. For example, you might have to create and change into a directory or load a module or a specific version of a software package before you call your *main* executable.

In RADICAL-Pilot this can be easily achieved by using `/bin/bash` as the executable in the *radical.pilot.ComputeUnitDescription* (page 63) and either pass a shell script directly as a string argument or transfer a shell script file as part of the ComputeUnit. The former works well for a small set of simple commands, while the second works best for more complex scripts.

Using `/bin/bash` as Executable

TODO - explain -c and -l

```
cu = radical.pilot.ComputeUnitDescription()
cu.executable  = "/bin/bash"
cu.arguments   = ["-l", "-c", " this && and && that """]
cu.cores       = 1
```

Using a Shell-Script File

TODO

```
TODO
```

1.5.6 Executing Multicore / Multithreaded ComputeUnits

Multithreaded Applications

MPI Applications

To define an MPI ComputeUnit, all you need to do is to set the `cores` and the `mpi` arguments in the ComputeUnitDescription.

```
pdesc = radical.pilot.ComputeUnitDescription()
[...]
pdesc.mpi      = True
pdesc.cores    = 32
```

```
import os
import sys
import radical.pilot as rp

# READ: The RADICAL-Pilot documentation:
#   http://radicalpilot.readthedocs.org/en/latest
#
# Try running this example with RADICAL_PILOT_VERBOSE=debug set if
# you want to see what happens behind the scenes!


#------------------------------------------------------------------------------
#
def pilot_state_cb (pilot, state) :
    """ this callback is invoked on all pilot state changes """

    print "[Callback]: ComputePilot '%s' state: %s." % (pilot.uid, state)

    if  state == rp.FAILED :
        sys.exit (1)


#------------------------------------------------------------------------------
#
def unit_state_cb (unit, state) :
    """ this callback is invoked on all unit state changes """
```

```
    print "[Callback]: ComputeUnit  '%s' state: %s." % (unit.uid, state)

    if  state == rp.FAILED :
        sys.exit (1)


# ------------------------------------------------------------------------------
#
if __name__ == "__main__":

    # Create a new session. A session is the 'root' object for all other
    # RADICAL-Pilot objects. It encapsulates the MongoDB connection(s).
    session = rp.Session()

    # Add a Pilot Manager. Pilot managers manage one or more ComputePilots.
    pmgr = rp.PilotManager(session=session)

    # Register our callback with the PilotManager. This callback will get
    # called every time any of the pilots managed by the PilotManager
    # change their state.
    pmgr.register_callback(pilot_state_cb)

    # Define a X-core on stamped that runs for N minutes and
    # uses $HOME/radical.pilot.sandbox as sandbox directoy.
    pdesc = rp.ComputePilotDescription()
    pdesc.resource = "xsede.stampede"
    pdesc.runtime  = 15 # N minutes
    pdesc.cores    = 16 # X cores
    pdesc.project  = "TG-MCB090174"

    # Launch the pilot.
    pilot = pmgr.submit_pilots(pdesc)

    cud_list = []

    for unit_count in range(0, 4):
        cu = rp.ComputeUnitDescription()
        cu.pre_exec      = ["module load python intel mvapich2 mpi4py"]
        cu.executable    = "python"
        cu.arguments     = ["helloworld_mpi.py"]
        cu.input_staging = ["helloworld_mpi.py"]

        # These two parameters are relevant to MPI execution:
        #   'cores' sets the number of cores required by the task
        #   'mpi' identifies the task as an MPI taskg
        cu.cores         = 8
        cu.mpi           = True


        cud_list.append(cu)

    # Combine the ComputePilot, the ComputeUnits and a scheduler via
    # a UnitManager object.
    umgr = rp.UnitManager(
        session=session,
        scheduler=rp.SCHED_DIRECT_SUBMISSION)

    # Register our callback with the UnitManager. This callback will get
```

```
    # called every time any of the units managed by the UnitManager
    # change their state.
    umgr.register_callback(unit_state_cb)

    # Add the previously created ComputePilot to the UnitManager.
    umgr.add_pilots(pilot)

    # Submit the previously created ComputeUnit descriptions to the
    # PilotManager. This will trigger the selected scheduler to start
    # assigning ComputeUnits to the ComputePilots.
    units = umgr.submit_units(cud_list)

    # Wait for all compute units to reach a terminal state (DONE or FAILED).
    umgr.wait_units()

    if not isinstance(units, list):
        units = [units]
    for unit in units:
        print "* Task %s - state: %s, exit code: %s, started: %s, finished: %s, stdout: %s" \
            % (unit.uid, unit.state, unit.exit_code, unit.start_time, unit.stop_time, unit.stdout)

    session.close()

# ------------------------------------------------------------------------------
```

This example uses this simple MPI4Py example as MPI executable (requires MPI4Py installed on the remote cluster):

```
#!/usr/bin/env python

# This is an example MPI4Py program that is used
# by different examples and tests.

import sys
import time
import traceback
from   mpi4py import MPI

try :
    print "start"
    SLEEP = 10
    name  = MPI.Get_processor_name()
    comm  = MPI.COMM_WORLD

    print "mpi rank %d/%d/%s"  % (comm.rank, comm.size, name)

    time.sleep(SLEEP)

    comm.Barrier()   # wait for everybody to synchronize here

except Exception as e :
    traceback.print_exc ()
    print "error : %s" % s
    sys.exit (1)

finally :
    print "done"
    sys.exit (0)
```

1.5.7 Application Control Flow with Callbacks

1.5.8 Mandelbrot set

Requirements

This Mandelbrot example needs the PIL library for both the "application side" and the "CU side". For the application side you need to install the Pillow module in the same virtual environment as you have installed RADICAL-Pilot into:

```
pip install Pillow
```

The examples are constructed in such a way that PIL is dynamically installed in the CU environment; more on that later.

Obtaining the code

Download the mandelbrot example via command line:

```
curl --insecure -Os https://raw.githubusercontent.com/radical-cybertools/radical.pilot/readthedocs/ex
curl --insecure -Os https://raw.githubusercontent.com/radical-cybertools/radical.pilot/readthedocs/ex
```

Customizing the example

Open the file mandelbrot_pilot_cores.py with your favorite editor. There is a critical section that must be filled in by the user. About halfway of this file it says, "BEGIN REQUIRED CU SETUP." This section defines the actual tasks to be executed by the pilot.

Let's discuss the above example. We define our executable as "python". Next, we need to provide the arguments. In this case, mandel_lines.py is the python script that creates parts of the mandelbrot fractal. The other arguments are the variables that the mandel_lines.py program needs in order to be executed. Note that this block of code is in a python for loop, therefore, e.g. "i" corresponds to what iteration we are on. This is a parallel code, the python uses as many cores as we define, (now we defined cores=4) to create smaller parts of the fractal simultaneously.

More About the Algorithm

This algorithm takes the takes the parameters of the Mandelbrot fractal and decompose the image into n different parts, where n is the number of the cores of the system. Then it runs for every part the mandelbrot Generator Code which is the mandel_lines.py. The mandel_lines.py creates n Images and then we compose the n images into one. The whole fractal Image. For every part of the image we create one Compute Unit.

Run the example

Save the file and executed:

```
python mandelbrot_pilot_cores.py 1024 1024 0 1024 0 1024
```

The parameters are the following: imgX, imgY: the dimensions of the mandelbrot image, e.g. 1024, 1024 xBeg, xEnd: the x-axis portion of the (sub-)image to calculate yBeg, yEnd: the y-axis portion of the (sub-)image to calculate

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 ...
...

All Compute Units completed successfully! Now..
Stitching together the whole fractal to : mandelbrot_full.gif
Images is now saved at the working directory..
Session closed, exiting now ...
```

When you finish the execution you may find the image in your working directory: mandelbrot_full.gif

1.5.9 K-Means Clustering

Introduction

This example implements the k-means algorithm using the RADICAL-Pilot API.

Obtaining the code

To download the source files of k-means algorithm:

```
curl --insecure -Os https://raw.githubusercontent.com/radical-cybertools/radical.pilot/readthedocs/ex
curl --insecure -Os https://raw.githubusercontent.com/radical-cybertools/radical.pilot/readthedocs/ex
curl --insecure -Os https://raw.githubusercontent.com/radical-cybertools/radical.pilot/readthedocs/ex
```

And to download an example dataset:

```
curl --insecure -Os https://raw.githubusercontent.com/radical-cybertools/radical.pilot/readthedocs/ex
```

Running the example

To give it a test drive try via command line the following command:

```
python k-means.py 3
```

where 3 is the number of clusters the user wants to create.

More About the Algorithm

This application creates the clusters of the elements found in the dataset4.in file. You can create your own file or create a new dataset file using the following generator:

```
curl --insecure -Os https://raw.githubusercontent.com/radical-cybertools/radical.pilot/readthedocs/ex
```

Run via command line:

```
python creating_dataset.py <number_of_elements>
```

The algorithm takes the elements from the dataset4.in file. Then, it chooses the first k centroids using the quickselect algorithm. It divides into number_of_cores files the initial file and pass each file as an argument to each Compute Unit. Every Compute Unit find in which cluster every element belongs to and creates k different sums of the elements coordinates. and returns this sum. Then, we sum all the sums of the CUs, and find the average elements who are closest to the new centroids. Afterwards, we do the same decomposition, but this time we try to find the new centroids. From each CU we find the nearest element to each centroid, and return them to the main program. Then we compare the results of all the CUs, and we decided who are the new centroids. If we have convergence we stop the algorithm, otherwise we start a new iteration.

1.6 Tutorial

This tutorial will walk you through the basic features of RADICAL-Pilot. Several examples illustrate some common usage patterns in distributed environments.

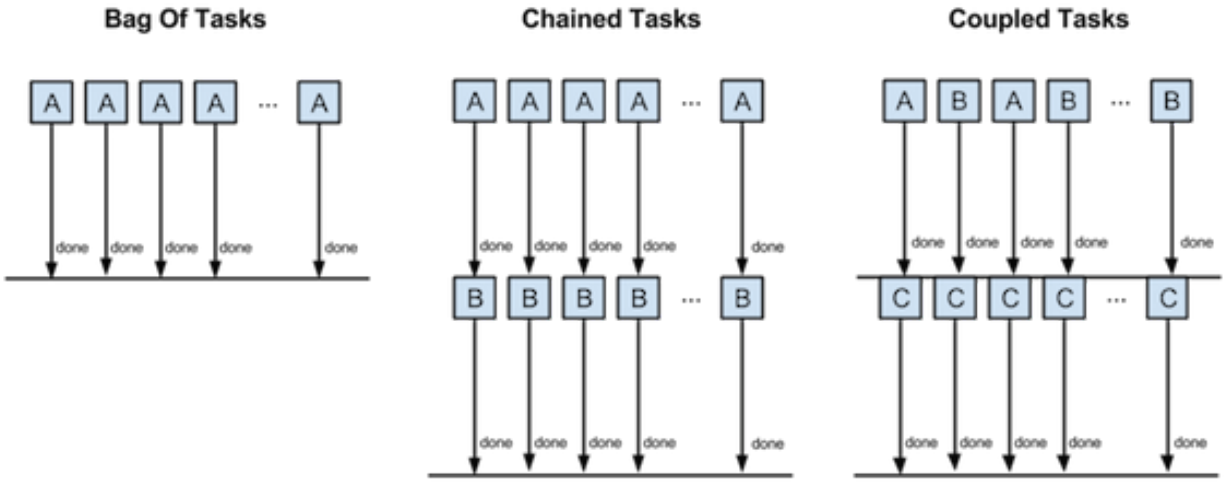
Prerequisites:

- You are familiar with Linux or UNIX.
- You can read and write Python code.
- You can use SSH and understand how public and private keys work.
- You understand the basic concepts of distributed computing.

You will learn how to:

- Submit multiple jobs through a 'pilot job' to your local workstation as well as a remote machine.
- Write programs that run multiple jobs concurrently, sequentially, or both, based on their requirements and dependencies.
- How to specify input files for tasks.
- MPI tasks are different from regular tasks.

Contents:



1.6.1 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') collectively, i.e. as one big job! Such usage arises for example to perform either a parameter sweep job or a set of ensemble simulation.

We will create an example which submits N jobs using RADICAL-Pilot. The jobs 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 and complete, you submit just one container job (called a Pilot) that reserves the number of cores needed to run all of your jobs. When this pilot becomes active, your tasks (which are named 'Compute Units' or 'CUs') are pulled by RADICAL-Pilot from the MongoDB server and executed.

Preparation

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

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:

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

```
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 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). For more details on logging, see under 'Debugging' in chapter *Developer Documentation* (page 48).

Give it a try with the above example:

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

1.6.2 Chained Tasks

What if you had two different executables to run? What if this second set of executables had some dependencies on data from A? Can you use one RADICAL-Pilot to run both 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 this as A is 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:

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:

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

```
python chained_tasks.py
```

1.6.3 Coupled Tasks

The script provides a simple workflow which submit a set of tasks A and tasks B and wait until they are completed and then submits 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:

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:

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

```
python coupled_tasks.py
```

1.6.4 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:

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:

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:

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
```

1.7 Unit Scheduler

1.7.1 Introduction

1.7.2 Direct Submission Scheduler (SCHED_DIRECT_SUBMISSION)

1.7.3 Round Robin Scheduler (SCHED_ROUND_ROBIN)

1.8 Testing

1.8.1 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.

1.8.2 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.

1.8.3 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://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
```

1.9 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.

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 536afe101d41c83696

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.

1.10 Frequently Asked Questions

1.10.1 Q: I see 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,
    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
pymongo.errors.OperationFailure: too many namespaces/collections
```

1.10.2 A: Try Cleaning-up MongoDB.

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.

1.10.3 Q: I see 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
```

1.10.4 A: Set up password-less, intra-node SSH access.

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)

1.10.5 Q: On Gordon I see "Failed to execvp() 'mybinary': No such file or directory (2)"

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 directo
```

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

1.10.7 Q: How many concurrent RADICAL-Pilot scripts can I execute?

A: 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.

1.10.8 Q: Why do I get errors from setuptools when trying to use a virtualenv?

1.10.9 A: 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.

1.11 Developer Documentation

1.11.1 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 git@github.com:radical-cybertools/radical.pilot.git
```

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

```
python setup.py install
```

1.11.2 License

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

1.11.3 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.

1.11.4 Debugging

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

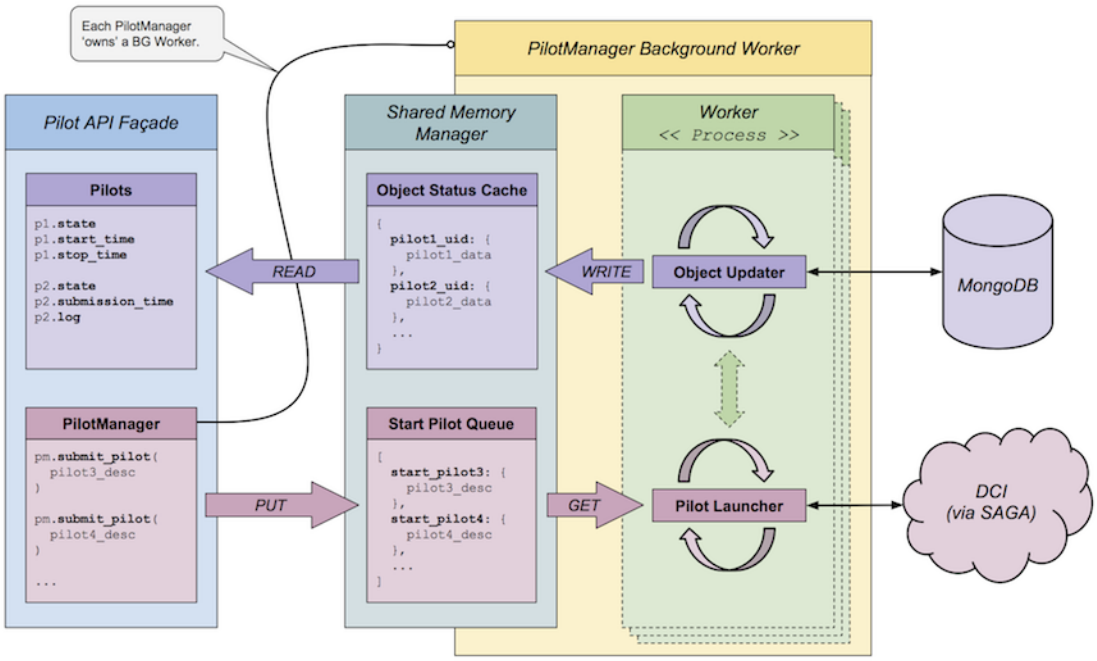
- *debug*
- *info*
- *warning*
- *error*

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.

1.11.5 RADICAL-Pilot Architecture

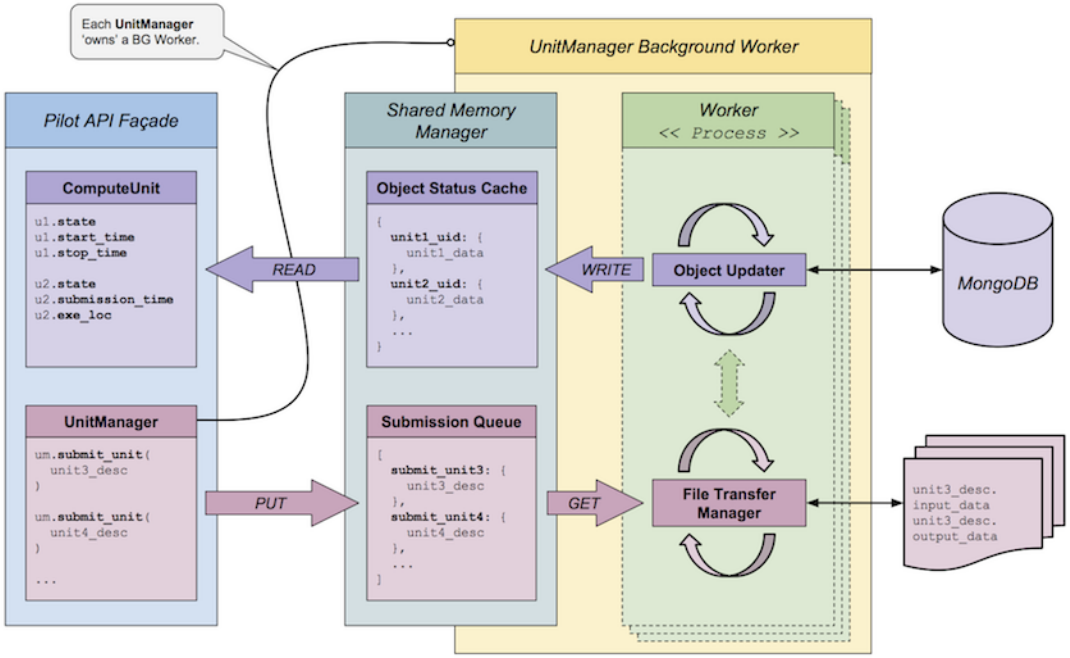
Describe architecture overview here.

PilotManager and PilotManager Worker



Download PDF version.

UnitManager and UnitManager Worker



Download PDF version.

1.12 API Reference

1.12.1 Sessions and Security Contexts

Sessions

class `radical.pilot.`**`Session`**(*database_url=None*, *database_name='radicalpilot'*, *session_uid=None*)

A Session encapsulates a RADICAL-Pilot instance and is the *root* object for all other RADICAL-Pilot objects.

A Session holds *radical.pilot.PilotManager* (page 53) and *radical.pilot.UnitManager* (page 59) instances which in turn hold `radical.pilot.Pilot` and *radical.pilot.ComputeUnit* (page 64) instances.

Each Session has a unique identifier `radical.pilot.Session.uid` that can be used to re-connect to a RADICAL-Pilot instance in the database.

Example:

```
s1 = radical.pilot.Session(database_url=DBURL)
s2 = radical.pilot.Session(database_url=DBURL, session_uid=s1.uid)

# s1 and s2 are pointing to the same session
assert s1.uid == s2.uid
```

`__init__`(*database_url=None*, *database_name='radicalpilot'*, *session_uid=None*)

Creates a new or reconnects to an exising session.

If called without a session_uid, a new Session instance is created and stored in the database. If session_uid is set, an existing session is retrieved from the database.

Arguments:

- **database_url** (*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.
- **database_name** (*string*): An alternative database name (default: 'radicalpilot').
- **session_uid** (*string*): If session_uid is set, we try re-connect to an existing session instead of creating a new one.

Returns:

- A new Session instance.

Raises:

- *radical.pilot.DatabaseError* (page 66)

`close`(*cleanup=True*, *terminate=True*, *delete=None*)

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.

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

list_pilot_managers()
Lists the unique identifiers of all *radical.pilot.PilotManager* (page 53) instances associated with this session.

Example:

```
s = radical.pilot.Session(database_url=DBURL)
for pm_uid in s.list_pilot_managers():
    pm = radical.pilot.PilotManager(session=s, pilot_manager_uid=pm_uid)
```

Returns:

- A list of *radical.pilot.PilotManager* (page 53) uids (*list* oif strings').

Raises:

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

get_pilot_managers(*pilot_manager_ids=None*)
Re-connects to and returns one or more existing PilotManager(s).

Arguments:

•**session** [*radical.pilot.Session* (page 51)]: The session instance to use.

•**pilot_manager_uid** [*string*]: The unique identifier of the PilotManager we want to re-connect to.

Returns:

•One or more new [*radical.pilot.PilotManager* (page 53)] objects.

Raises:

•radical.pilot.pilotException if a PilotManager with *pilot_manager_uid* doesn't exist in the database.

list_unit_managers()
Lists the unique identifiers of all *radical.pilot.UnitManager* (page 59) instances associated with this session.

Example:

```
s = radical.pilot.Session(database_url=DBURL)
for pm_uid in s.list_unit_managers():
    pm = radical.pilot.PilotManager(session=s, pilot_manager_uid=pm_uid)
```

Returns:

- A list of *radical.pilot.UnitManager* (page 59) uids (*list* of *strings*).

Raises:

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

get_unit_managers(*unit_manager_ids=None*)
Re-connects to and returns one or more existing UnitManager(s).

Arguments:

•**session** [*radical.pilot.Session* (page 51)]: The session instance to use.

•**pilot_manager_uid** [*string*]: The unique identifier of the PilotManager we want to re-connect to.

Returns:

•One or more new [*radical.pilot.PilotManager* (page 53)] objects.

Raises:

•`radical.pilot.pilotException` if a PilotManager with *pilot_manager_uid* doesn't exist in the database.

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
rc.name                 = "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_key*)
Returns a dictionary of the requested resource config

Security Contexts

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

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

1.12.2 Pilots and PilotManagers

PilotManagers

class `radical.pilot.`**PilotManager**(*session*, *pilot_launcher_workers=1*, *_reconnect=False*)
A PilotManager holds *radical.pilot.ComputePilot* (page 57) instances that are submitted via the *radical.pilot.PilotManager.submit_pilots()* (page 55) method.

It is possible to attach one or more *Using Local and Remote HPC Resources* (page 5) to a PilotManager to outsource machine specific configuration parameters to an external configuration file.

Each PilotManager has a unique identifier `radical.pilot.PilotManager.uid` that can be used to re-connect to previoulsy created PilotManager in a given *radical.pilot.Session* (page 51).

Example:

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

pm1 = radical.pilot.PilotManager(session=s, resource_configurations=RESCONF)
# Re-connect via the 'get()' method.
pm2 = radical.pilot.PilotManager.get(session=s, pilot_manager_id=pm1.uid)

# pm1 and pm2 are pointing to the same PilotManager
assert pm1.uid == pm2.uid
```

__init__(*session*, *pilot_launcher_workers=1*, *_reconnect=False*)

Creates a new PilotManager and attaches is to the session.

Note: The *resource_configurations* (see *Using Local and Remote HPC Resources* (page 5)) parameter is currently mandatory for creating a new PilotManager instance.

Arguments:

- **session** [*radical.pilot.Session* (page 51)]: The session instance to use.
- **resource_configurations** [*string* or *list of strings*]: A list of URLs pointing to *Using Local and Remote HPC Resources* (page 5). Currently *file://*, *http://* and *https://* URLs are supported.

  If one or more resource_configurations are provided, Pilots submitted via this PilotManager can access the configuration entries in the files via the `ComputePilotDescription`. For example:

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

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

pilot = pm.submit_pilots(pd)
```

- pilot_launcher_workers (*int*): The number of pilot launcher worker processes to start in the background.

Note: *pilot_launcher_workers* can be used to tune RADICAL-Pilot's performance. However, you should only change the default values if you know what you are doing.

Returns:

- A new *PilotManager* object [*radical.pilot.PilotManager* (page 53)].

Raises:

- *radical.pilot.PilotException* (page 66)

close(*terminate=True*)

Shuts down the PilotManager and its background workers in a coordinated fashion.

Arguments:

•**terminate** [*bool*]: If set to True, all active pilots will get canceled (default: False).

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

submit_pilots(*pilot_descriptions*)
Submits a new *radical.pilot.ComputePilot* (page 57) to a resource.

Returns:

•One or more *radical.pilot.ComputePilot* (page 57) instances [*list of :class:'radical.pilot.ComputePilot*].

Raises:

•*radical.pilot.PilotException* (page 66)

list_pilots()
Lists the unique identifiers of all *radical.pilot.ComputePilot* (page 57) instances associated with this PilotManager

Returns:

•A list of *radical.pilot.ComputePilot* (page 57) uids [*string*].

Raises:

•*radical.pilot.PilotException* (page 66)

get_pilots(*pilot_ids=None*)
Returns one or more *radical.pilot.ComputePilot* (page 57) instances.

Arguments:

•**pilot_uids** [*list of strings*]: If pilot_uids is set, only the Pilots with the specified uids are returned. If pilot_uids is *None*, all Pilots are returned.

Returns:

•A list of *radical.pilot.ComputePilot* (page 57) objects [*list of :class:'radical.pilot.ComputePilot*].

Raises:

•*radical.pilot.PilotException* (page 66)

wait_pilots(*pilot_ids=None, state=['Done', 'Failed', 'Canceled'], timeout=None*)
Returns when one or more `radical.pilot.ComputePilots` reach a specific state or when an optional timeout is reached.

If *pilot_uids* is *None*, *wait_pilots* returns when **all** Pilots reach the state defined in *state*.

Arguments:

•**pilot_uids** [*string* or *list of strings*] If pilot_uids is set, only the Pilots with the specified uids are considered. If pilot_uids is *None* (default), all Pilots are considered.

•**state** [*list of strings*] The state(s) that Pilots have to reach in order for the call to return.

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

–`radical.pilot.DONE`

–`radical.pilot.FAILED`

–`radical.pilot.CANCELED`

•**timeout** [*float*] Optional timeout in seconds before the call returns regardless whether the Pilots have reached the desired state or not. The default value **-1.0** never times out.

Raises:

•*radical.pilot.PilotException* (page 66)

cancel_pilots(*pilot_ids=None*)

Cancels one or more ComputePilots.

Arguments:

•**pilot_uids** [*string* or *list of strings*] If pilot_uids is set, only the Pilots with the specified uids are canceled. If pilot_uids is *None*, all Pilots are canceled.

Raises:

•*radical.pilot.PilotException* (page 66)

register_callback(*callback_function*, *callback_data=None*)

Registers a new callback function with the PilotManager. Manager-level callbacks get called if any of the ComputePilots managed by the PilotManager change their state.

All callback functions need to have the same signature:

```
def callback_func(obj, state, data)
```

where `object` is a handle to the object that triggered the callback, `state` is the new state of that object, and `data` are the data passed on callback registration.

ComputePilotDescription

class radical.pilot.**ComputePilotDescription**

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 55) to instantiate a new pilot.

Note: A ComputePilotDescription **MUST** define at least *resource* (page 56) and the number of *cores* (page 57) to allocate on the target resource.

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 5) 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 55). 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 5). 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.

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 56)) 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 56)), defining *project* will override it explicitly.

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**

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 55) has to be used instead.

Example:

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

pd = radical.pilot.ComputePilotDescription()
pd.resource = "local.localhost"
pd.cores    = 2
pd.runtime  = 5 # minutes

pilot = pm.submit_pilots(pd)
```

---

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

uid
Returns the Pilot's unique identifier.

The uid identifies the Pilot within the `PilotManager` and can be used to retrieve an existing Pilot.

Returns:

- A unique identifier (string).

description
Returns the pilot description the pilot was started with.

sandbox
Returns the Pilot's 'sandbox' / working directory url.

Returns:

- A URL string.

state
Returns the current state of the pilot.

state_history
Returns the complete state history of the pilot.

stdout
Returns the stdout of the pilot.

stderr
Returns the stderr of the pilot.

logfile
Returns the logfile of the pilot.

log
Returns the log of the pilot.

resource_detail
Returns the names of the nodes managed by the pilot.

pilot_manager
Returns the pilot manager object for this pilot.

unit_managers
Returns the unit manager object UIDs for this pilot.

units
Returns the units scheduled for this pilot.

submission_time
Returns the time the pilot was submitted.

start_time
Returns the time the pilot was started on the backend.

stop_time
Returns the time the pilot was stopped.

resource
Returns the resource.

register_callback(*callback_func*, *callback_data=None*)
Registers a callback function that is triggered every time the ComputePilot's state changes.

All callback functions need to have the same signature:

```
def callback_func(obj, state, data)
```

where `object` is a handle to the object that triggered the callback, `state` is the new state of that object, and `data` is the data passed on callback registration.

wait (*state=['Done', 'Failed', 'Canceled'], 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 **terminal** 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.

Raises:

- `radical.pilot.exceptions.radical.pilotException` if the state of the pilot cannot be determined.

cancel ()
Sends sends a termination request to the pilot.

Raises:

- `radical.pilot.radical.pilotException` if the termination request cannot be fulfilled.

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

1.12.3 ComputeUnits and UnitManagers

UnitManager

class `radical.pilot.`**UnitManager** (*session, scheduler=None, input_transfer_workers=2, output_transfer_workers=2, _reconnect=False*)
A UnitManager manages *radical.pilot.ComputeUnit* (page 64) 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` gets executed on which `Pilot`.

Each UnitManager has a unique identifier *radical.pilot.UnitManager.uid* (page 60) that can be used to re-connect to previoulsy created UnitManager in a given *radical.pilot.Session* (page 51).

Example:

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

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

pd = radical.pilot.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 = 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.SCHED_ROUND_ROBIN)
um.add_pilot(p1)
um.submit_units(compute_units)
```

__init__(*session*, *scheduler=None*, *input_transfer_workers=2*, *output_transfer_workers=2*, *_reconnect=False*)

Creates a new UnitManager and attaches it to the session.

Args:

- session (*string*): The session instance to use.
- scheduler (*string*): The name of the scheduler plug-in to use.
- input_transfer_workers (*int*): The number of input file transfer worker processes to launch in the background.
- output_transfer_workers (*int*): The number of output file transfer worker processes to launch in the background.

Note: *input_transfer_workers* and *output_transfer_workers* can be used to tune RADICAL-Pilot's file transfer performance. However, you should only change the default values if you know what you are doing.

Raises:

- *radical.pilot.PilotException* (page 66)

close()

Shuts down the UnitManager and its background workers in a coordinated fashion.

as_dict()

Returns a Python dictionary representation of the UnitManager object.

uid

Returns the unique id.

scheduler

Returns the scheduler name.

scheduler_details

Returns the scheduler logs.

add_pilots(*pilots*)

Associates one or more pilots with the unit manager.

Arguments:

•**pilots** [*radical.pilot.ComputePilot* (page 57) or list of *radical.pilot.ComputePilot* (page 57)]: The pilot objects that will be added to the unit manager.

Raises:

•*radical.pilot.PilotException* (page 66)

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

Returns:

•A list of *radical.pilot.ComputePilot* (page 57) UIDs [*string*].

Raises:

•*radical.pilot.PilotException* (page 66)

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

Returns:

•A list of *radical.pilot.ComputePilot* (page 57) instances.

Raises:

•*radical.pilot.PilotException* (page 66)

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

TODO: Implement 'drain'.

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 *ACTIVE* units will be canceled.

Raises:

•*radical.pilot.PilotException* (page 66)

list_units()
Returns the UIDs of the *radical.pilot.ComputeUnit* (page 64) managed by this unit manager.

Returns:

•A list of *radical.pilot.ComputeUnit* (page 64) UIDs [*string*].

submit_units(*unit_descriptions*)
Submits on or more *radical.pilot.ComputeUnit* (page 64) instances to the unit manager.

Arguments:

•**unit_descriptions** [*radical.pilot.ComputeUnitDescription* (page 63) or list of *radical.pilot.ComputeUnitDescription* (page 63)]: The description of the compute unit instance(s) to create.

Returns:

•A list of *radical.pilot.ComputeUnit* (page 64) objects.

Raises:

- *radical.pilot.PilotException* (page 66)

get_units(*unit_ids=None*)

Returns one or more compute units identified by their IDs.

Arguments:

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

Returns:

- A list of *radical.pilot.ComputeUnit* (page 64) objects.

Raises:

- *radical.pilot.PilotException* (page 66)

wait_units(*unit_ids=None, state=['Done', 'Failed', 'Canceled'], timeout=None*)

Returns when one or more `radical.pilot.ComputeUnits` reach a specific state.

If *unit_uids* is *None*, *wait_units* returns when **all** ComputeUnits reach the state defined in *state*.

Example:

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

Arguments:

- **unit_uids** [*string* or *list of strings*] If unit_uids is set, only the ComputeUnits with the specified uids are considered. If unit_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.DONE`
  - `radical.pilot.FAILED`
  - `radical.pilot.CANCELED`
- **timeout** [*float*] Timeout in seconds before the call returns regardless of Pilot state changes. The default value **None** waits forever.

Raises:

- *radical.pilot.PilotException* (page 66)

cancel_units(*unit_ids=None*)

Cancel one or more `radical.pilot.ComputeUnits`.

Arguments:

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

Raises:

- *radical.pilot.PilotException* (page 66)

register_callback(*callback_function, metric='UNIT_STATE', callback_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 callback_func(obj, value, 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`**

A ComputeUnitDescription object describes the requirements and properties of a *radical.pilot.ComputeUnit* (page 64) and is passed as a parameter to *radical.pilot.UnitManager.submit_units()* (page 61) to instantiate and run a new ComputeUnit.

Note: A ComputeUnitDescription **MUST** define at least an *executable* (page 63).

Example:

```
# TODO
```

`executable`
(*Attribute*) The executable to launch (*string*) [*mandatory*].

`cores`
(*Attribute*) The number of cores (int) required by the executable. (int) [*mandatory*].

`mpi`
(*Attribute*) Set to true if the task is an MPI task. (bool) [*optional*].

`name`
(*Attribute*) A descriptive name for the compute unit (*string*) [*optional*].

`arguments`
(*Attribute*) The arguments for *executable* (page 63) (*list* of *strings*) [*optional*].

`environment`
(*Attribute*) Environment variables to set in the execution environment (*dict*) [*optional*].

`stdout`
(*Attribute*) the name of the file to store stdout in.

`stderr`
(*Attribute*) the name of the file to store stderr in.

`input_staging`
(*Attribute*) The files that need to be staged before execution (*list* of *staging directives*) [*optional*].

Note: TODO: Explain input staging.

`output_staging`
(*Attribute*) The files that need to be staged after execution (*list* of *staging directives*) [*optional*].

Note: TODO: Explain output staging.

pre_exec
(*Attribute*) Actions to perform before this task starts (*list* of *strings*) [*optional*].

post_exec
(*Attribute*) Actions to perform after this task finishes (*list* of *strings*) [*optional*].

Note: Before the BigBang, there was nothing ...

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

Note: TODO: explain in detal, reference ENMDTK.

restartable
(*Attribute*) 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)

Note: TODO: explain in detal, reference ENMDTK.

cleanup
[Type: *bool*] [optional] If cleanup 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.

ComputeUnit

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

Note: A ComputeUnit cannot be created directly. The factory method *radical.pilot.UnitManager.submit_units()* (page 61) has to be used instead.

Example:

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

ud = radical.pilot.ComputeUnitDescription()
ud.executable = "/bin/date"
ud.cores      = 1

unit = umgr.submit_units(ud)
```

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

uid
Returns the unit's unique identifier.

The uid identifies the ComputeUnit within a UnitManager and can be used to retrieve an existing ComputeUnit.

Returns:

- A unique identifier (string).

name
Returns the unit's application specified name.

Returns:

- A name (string).

working_directory
Returns the full working directory URL of this ComputeUnit.

pilot_id
Returns the pilot_id of this ComputeUnit.

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

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

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

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

description
Returns the ComputeUnitDescription the ComputeUnit was started with.

state
Returns the current state of the ComputeUnit.

state_history
Returns the complete state history of the ComputeUnit.

exit_code
Returns the exit code of the ComputeUnit.

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

log
Returns the logs of the ComputeUnit.

execution_details
Returns the exeuction location(s) of the ComputeUnit.

execution_locations
Returns the exeuction location(s) of the ComputeUnit. This is just an alias for execution_details.

submission_time
Returns the time the ComputeUnit was submitted.

start_time
Returns the time the ComputeUnit was started on the backend.

stop_time
Returns the time the ComputeUnit was stopped.

register_callback(*callback_func*, *callback_data=None*)
Registers a callback function that is triggered every time the ComputeUnit's state changes.

All callback functions need to have the same signature:

```
def callback_func(obj, state)
```

where `object` is a handle to the object that triggered the callback and `state` is the new state of that object.

wait(*state=['Done', 'Failed', 'Canceled'], timeout=None*)
Returns when the ComputeUnit reaches a specific state or when an optional timeout is reached.

Arguments:

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

  By default *wait* waits for the compute unit to reach a **terminal** 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 compute unit has reached the desired state or not. The default value **None** never times out.

Raises:

cancel()
Cancel the ComputeUnit.

Raises:

- `radical.pilot.radical.pilotException`

1.12.4 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()`.

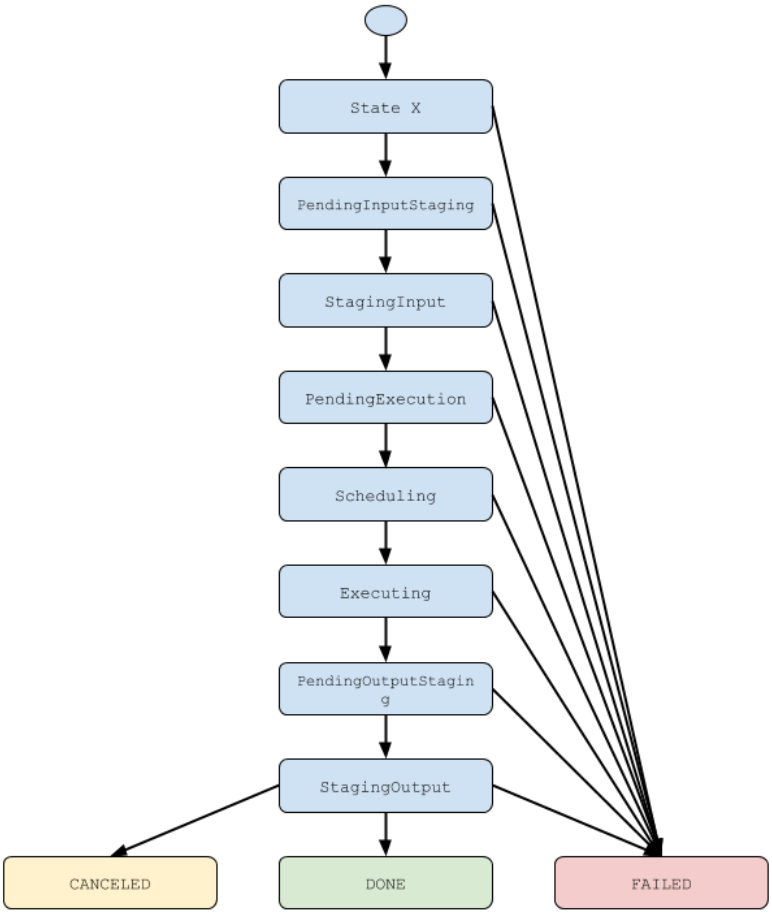
get_object()
Return the object instance on whose activity the exception was raised.

get_message()
Return the error message associated with the exception

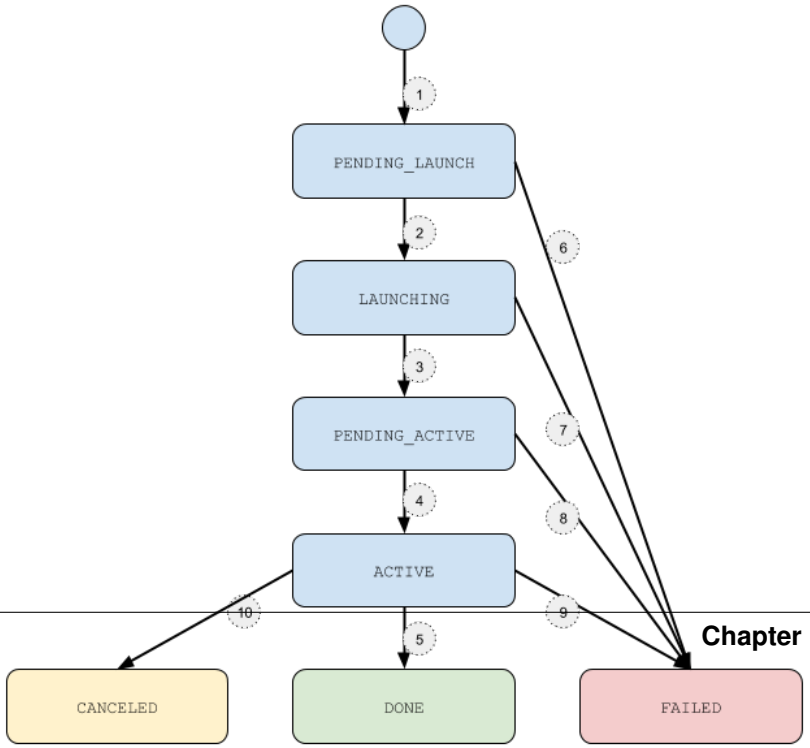
class `radical.pilot.`**DatabaseError**(*msg*, *obj=None*)
TODO: Document me!

1.12.5 State Models

ComputeUnit State Model



ComputePilot State Model



1. A new compute pilot is launched via *radical.pilot.PilotManager.submit_pilots()* (page 55)
2. The pilot is submitted to the remote resource and enters LAUNCHING state.
3. The pilot has been succesfully launched on the remote machine and is now waiting to become ACTIVE.
4. The pilot has been launched by the queueing system and is now in ACTIVE STATE.
5. The pilot has finished execution regularly and enters DONE state.
6. An error has occured during preparation for pilot launching and the pilot enters FAILED state.
7. An error has occured during pilot launching and the pilot enters FAILED state.
8. An error has occured on the backend and the pilot couldn't become active and the pilot enters FAILED state.
9. An error has occured during pilot runtime and the pilot enters FAILED state.
10. The active pilot has been canceled via the *radical.pilot.ComputePilot.cancel()* (page 59) call and enters CANCELED state.

1.13 Release Notes

- For a list of bug fixes, see https://github.com/radical-cybertools/radical.pilot/issues?q=is%3Aissue+is%3Aclosed+sort%3Aupdated-desc
- For a list of open issues and known problems, see https://github.com/radical-cybertools/radical.pilot/issues?q=is%3Aissue+is%3Aopen+

1.13.1 0.23 Release 2014-12-13

- fix #455

1.13.2 0.22 Release 2014-12-11

- several state races fixed
- fix to tools for session cleanup and purging
- partial fix for pilot cancelation
- improved shutdown behavior
- improved hopper support
- adapt plotting to changed slothistory format
- make instructions clearer on data staging examples
- addresses issue #216
- be more resilient on pilot shutdown
- take care of cancelling of active pilots
- fix logic error on state check for pilot cancellation
- fix blacklight config (#360)
- attempt to cancel pilots timely...
- as fallback, use PPN information provided by SAGA

- hopper usues torque (thanks Mark!)
- Re-fix blacklight config. Addresses #359 (again).
- allow to pass application data to callbacks
- threads should not be daemons...
- workaround on failing bson encoding...
- report pilot id on cu inspection
- ignore caching errors
- also use staging flags on input staging
- stampede environment fix
- Added missing stampede alias
- adds timestamps to unit and pilot logentries
- fix state tags for plots
- fix plot style for waitq
- introduce UNSCHEDULED state as per #233
- selectable terminal type for plot
- document pilot log env
- add faq about VE problems on setuptools upgrade
- allow to specify session cache files
- added configuration for BlueBiou (Thanks Jordane)
- better support for json/bson/timestamp handling; cache mongodb data for stats, plots etc
- localize numpy dependency
- retire input_data and output_data
- remove obsolete staging examples
- address #410
- fix another subtle state race

1.13.3 0.21 Release 2014-10-29

- Documentation of MPI support
- Documentation of data staging operations
- correct handling of data transfer exceptions
- fix handling of non-ascii data in unit stdio
- simplify switching of access schemas on pilot submission
- disable pilot virtualenv for unit execution
- MPI support for DaVinci
- performance optimizations on file transfers, unit sandbox setup
- fix ibrun tmp file problem on stampede

1.13.4 0.19 Release September 12. 2014

- The Milestone 8 release (MS.8)
- Closed Tickets:
  - https://github.com/radical-cybertools/radical.pilot/issues?q=is%3Aclosed+milestone%3AMS-8+

1.13.5 0.18 Release July 22. 2014

- The Milestone 7 release (MS.7)
- Closed Tickets:
  - https://github.com/radical-cybertools/radical.pilot/issues?milestone=13&state=closed

1.13.6 0.17 Release June 18. 2014

Bugfix release - fixed file permissions et al. :/

1.13.7 0.16 Release June 17. 2014

Bugfix release - fixed file permissions et al.

1.13.8 0.15 Release June 12. 2014

Bugfix release - fixed distribution MANIFEST:

https://github.com/radical-cybertools/radical.pilot/issues/174

1.13.9 0.14 Release June 11. 2014

Closed Tickets:

- https://github.com/radical-cybertools/radical.pilot/issues?milestone=16&state=closed

New Features

- Experimental pilot-agent for Cray systems
- New multi-core agent with MPI support
- New ResourceConfig mechanism does not reuquire the user to add

resource configurations explicitly. Resources can be configured programatically on API-level.

API Changes:

- ComputeUnitDescription.working_dir_priv removed
- Extended state model
- resource_configurations parameter removed from PilotManager c'tor

1.13.10 0.13 Release May 19. 2014

- ExTASY demo release
- Support for project / allocation
- Updated / simplified resource files
- Refactored bootstrap mechnism

1.13.11 0.12 Release May 09. 2014

- Updated resource files
- Updated state model
- Closed tickets: - https://github.com/radical-cybertools/radical.pilot/issues?milestone=12&state=closed

1.13.12 0.11 Release Apr. 29. 2014

- Fixes error in state history reporting

1.13.13 0.10 Release Apr. 29. 2014

- Support for state transition introspection via CU/Pilot state_history
- Cleaned up an streamlined Input and Outpout file transfer workers
- Support for interchangeable pilot agents
- Closed tickets: - https://github.com/radical-cybertools/radical.pilot/issues?milestone=11&state=closed

1.13.14 0.9 Release Apr. 16. 2014

- Support for output file staging
- Streamlines data model
- More loosely coupled components connected via DB queues
- Closed tickets: - https://github.com/radical-cybertools/radical.pilot/issues?milestone=10&state=closed

1.13.15 0.8 Release Mar. 24. 2014

- Renamed codebase from sagapilot to radical.pilot
- Added explicit close() calls to PM, UM and Session.
- Cloesed tickets: - https://github.com/radical-cybertools/radical.pilot/issues?milestone=9&state=closed

1.13.16 0.7 Release Feb. 25. 2014

- Added support for callbacks
- Added support for input file transfer !
- Closed tickets: - https://github.com/radical-cybertools/radical.pilot/issues?milestone=8&state=closed

1.13.17 0.6 Release Feb. 24. 2014

- BROKEN RELEASE

1.13.18 0.5 Release Feb. 06. 2014

- Tutorial 2 release (Github only)
- Added support for multiprocessing worker
- Support for CU stdout and stderr transfer via MongoDB GridFS
- Closed tickets: - https://github.com/saga-project/saga-pilot/issues?milestone=7&page=1&state=closed

1.13.19 0.4 Release

- Tutorial 1 release (Github only)
- Consistent naming (sagapilot instead of sinon)

1.13.20 0.1.3 Release

- Github only release:

pip install –upgrade -e git://github.com/saga-project/saga-pilot.git@master#egg=saga-pilot

- Added logging
- Added security context handling
- Closed tickets: - https://github.com/saga-project/saga-pilot/issues?milestone=3&state=closed

1.13.21 0.1.2 Release

- Github only release:

pip install –upgrade -e git://github.com/saga-project/saga-pilot.git@master#egg=saga-pilot

- Closed tickets: - https://github.com/saga-project/saga-pilot/issues?milestone=4&state=closed

Symbols

A

C

D

E

F

G

L

N

P

R

S

U

W