Chiminey Documentation

Release 1.00

Ian Thomas

June 09, 2016

Note: This documentation is under construction!

Contents:

Chiminey Installation Guide

This document describes how to install a Chiminey platform via Docker, which is an automatic software deployment tool.

1.1 Requirements

Docker 1.7+ is needed. Follow the links below to install docker on your machine.

- *Mac OS X and Windows*
- *Linux*

1.1.1 Mac OS X and Windows

Here, we create a virtual machine that runs docker.

1. Download Docker Toolbox from https://www.docker.com/toolbox.

2. When the download is complete, open the installation dialog box by double-clicking the downloaded file.

3. Follow the on-screen prompts to install the Docker toolbox. You may be prompted for password just before the installation begins. You need to enter your password to continue.

4. When the installation is completed, press `Close` to exit.

5. Verify that `docker-engine` and `docker-compose` are installed correctly.

   - Open Docker Quickstart Terminal from your application folder. The resulting output looks like the following:

   - Run docker engine:

   ```
   $ docker run hello-world
   ```

   – You will see a message similar to the one below:

   ```
   Unable to find image 'hello-world:latest' locally
   latest: Pulling from library/hello-world
   03f4658f8b78: Pull complete
   a3ed95caeb02: Pull complete
   Digest: sha256:8be990ef2aeb16dbcb92...
   Status: Downloaded newer image for hello-world:latest
   Hello from Docker.
   ```
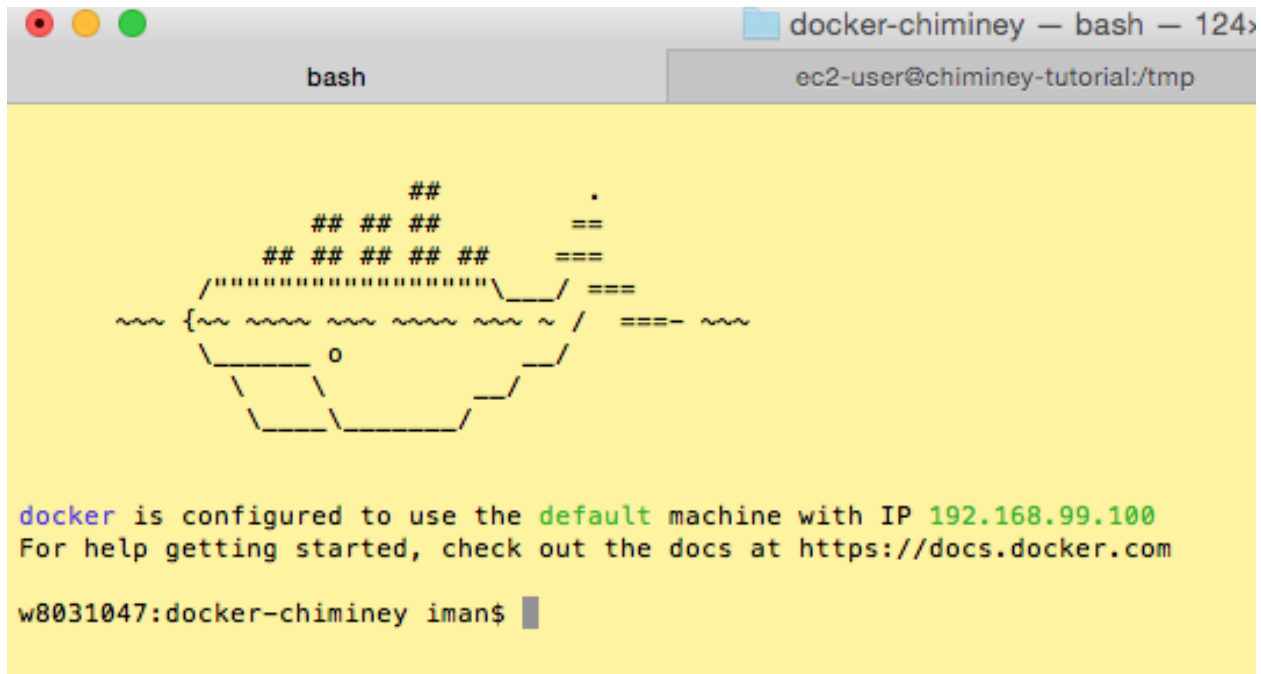
Fig. 1.1: Figure. Docker Virtual Machine on Mac OS X or Windows

```
        This message shows that your installation appears to be
            working correctly.
    ...
```

- Run docker-compose:

```
$ docker-compose --version
```

- – The output will be docker-compose version x.x.x, build xxxxxxx
- – For users with an older Mac, you will get Illegal instruction: 4. This error can be fixed by upgrading docker-compose:

```
$ pip install --upgrade docker-compose
```

1.1.2 Linux

Docker, specifically docker-engine and docker-compose, needs to be installed directly on your linux-based OS. Refer to the Docker online documentation to install the two packages:

1. Install docker-engine
2. Install docker-compose

1.2 Installation

1. For Mac OS X and Windows users, open *Docker Quickstart Terminal*. For linux-based OS users, login to your machine and open a terminal.

2. Check if `git` is installed. Type `git` on your terminal.

   • If git is installed, the following message will be shown:

```
usage: git [--version] [--help] [-C <path>] ..
           [--exec-path[=<path>]] [--html-path] [...
           [-p|--paginate|--no-pager] [--no- ...
           [--git-dir=<path>] [--work-tree=<path>]...
           <command> [<args>]
           ...
```

   • If git is not installed, you will see `git:  command not found`. Download and install `git` from
     http://git-scm.com/download

3. Clone the `docker-chiminey` source code from http://github.com.au:

```
$ git clone https://github.com/chiminey/docker-chiminey.git
```

4. Change your working directory:

```
$ cd docker-chiminey
```

5. Setup a self-signed certificate. You will be prompted to enter country code, state, city, and etc:

```
$ sh makecert
```

6. Deploy the Chiminey platform:

```
$ docker-compose up -d
```

7. Verify Chiminey was deployed successfully.

   • Retrieve the IP address of your machine

     – For Mac and Windows users, type `env | grep DOCKER_HOST`. The expected output has a format `DOCKER_HOST=tcp://IP:port`, for example. `DOCKER_HOST=tcp://192.168.99.100:2376`. Thus, your IP address is 192.168.99.100.

     – For linux users, the command `ifconfig` prints your our machine's IP address.

   • Open a browser and visit the Chiminey portal at IP, in our example, http://192.168.99.100. After a while, the Chiminey portal will be shown.

1.3 Configuration

Here, we will configure the Chiminey deployment by creating a superuser, initialising the database, and signing up a regular user.

1. For Mac OS X and Windows users, open *Docker Quickstart Terminal*. For linux-based OS users, login to your machine and open a terminal.

2. Change to `docker-chiminey` directory:

```
$ cd docker-chiminey
```

3. Create a superuser:

```
$ ./createsuper
```

4. Initialise the database:
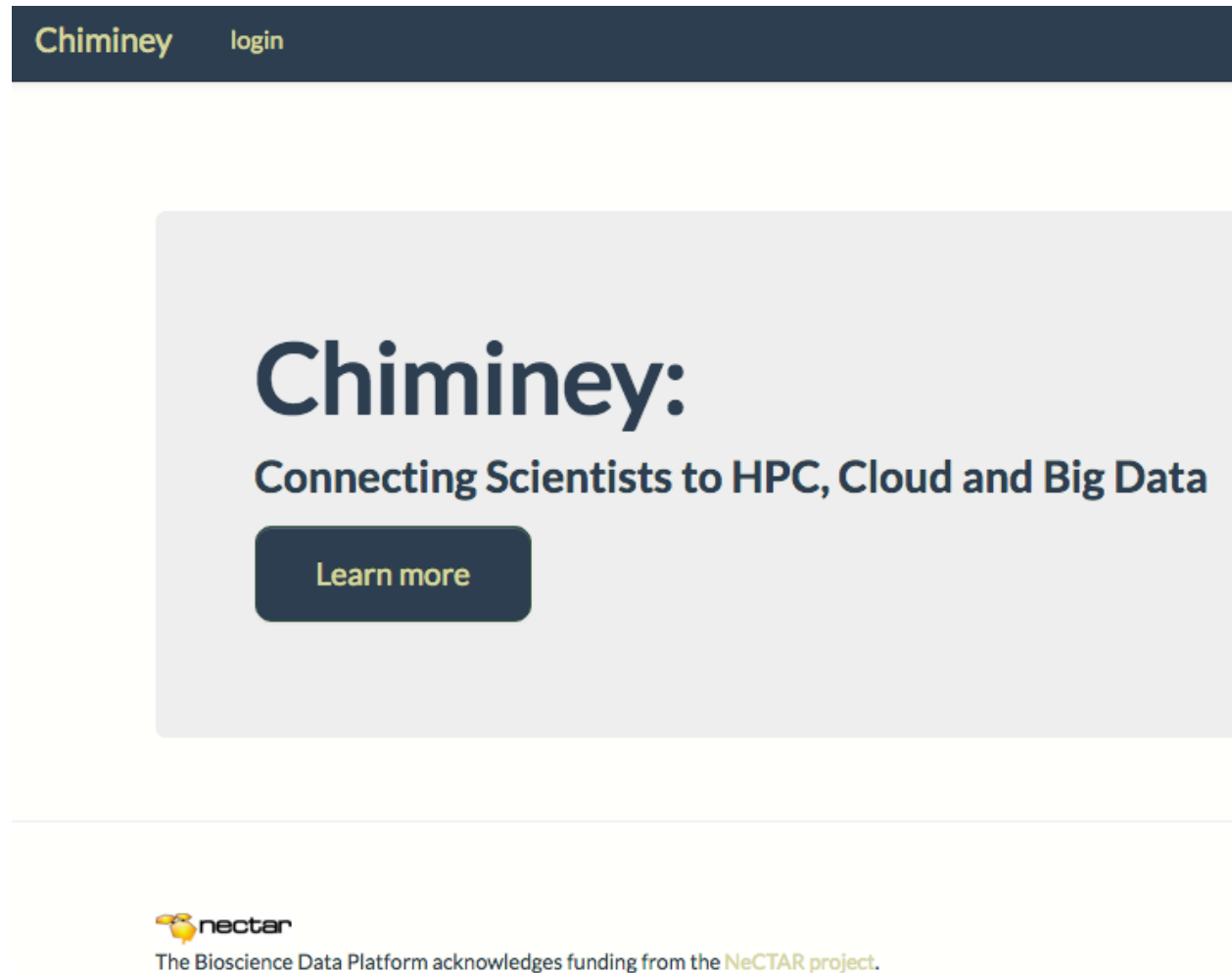
Fig. 1.2: Figure. Chiminey Portal

```
$ ./init
```

5. Create a regular user:

```
$ ./createuser
```

6. Verify the Chiminey platform is configured correctly.

- Open a browser and visit the Chiminey portal.

- Login with your regular username and password. After successful login, you will be redirected to a webpage that displays a list of jobs. Since no jobs are run yet, the list is empty.

1.4 Smart Connectors Activation

When a Chiminey platform is deployed, each *smart connector <smart_connector_desc>*, which is the core concept within Chiminey that enables endusers to perform complex computations on distributed computing facilities with minimal effort, needs to be explicitly activated.

1. For Mac OS X and Windows users, open *Docker Quickstart Terminal*. For linux-based OS users, login to your machine and open a terminal.

2. Change to `docker-chiminey` directory:

```
$ cd docker-chiminey
```

3. List all available smart connectors:

```
$ ./listsc

NAME:        DESCRIPTION
hrmclite:    Hybrid Reverse Monte Carlo without PSD
randnum:     Randnum generator, with timestamp
wordcount:   Counting words via Hadoop
```

4. Activate a smart connector. The syntax to activate a smart connector is *./activatesc smart-connector-name*. Thus, activate *randnum* smart connector as follows:

```
$ ./activatesc randnum
```

5. Verify the smart connector is successfully activated.

- Open a browser and visit the Chiminey portal.

- Login with your regular username and password.

- Click Create Job. *randnum* will appear under the Smart Connectors list.

See also:

https://www.djangoproject.com/ The Django Project

https://docs.djangoproject.com/en/1.4/intro/install/ Django Quick Install Guide

Enduser Manual

An end-user submits a smart connector job, monitors the job, visualises and curates job results. In this documentation, following topics are covered:

2.1 Getting Chiminey Account

Chiminey accounts are managed by admin users. Therefore, in order to get access to a specific Chiminey deployment, the end-user should contact the admin of the deployed Chiminey platform.

2.1.1 Login

End-users login via the web interface of the Chiminey platform

1. Click *login* on the home page

2. Enter Chiminey credentials

3. Click *Login*

2.1.2 Logout

1. click *Logout*

2.2 Resource Management

A Chiminey platform supports access to computation and storage resources. A computation resource is where the core functionality of a smart connector is executed while a storage resource is used to retrieve input files and store output files. Prior to submitting a job, end-users need to register at least one computation and one storage resources. In this section, following topics are covered:

- *Registering Compute Resources*
- *Registering Storage Resources*
- *Updating Resources*
- *Removing Resources*
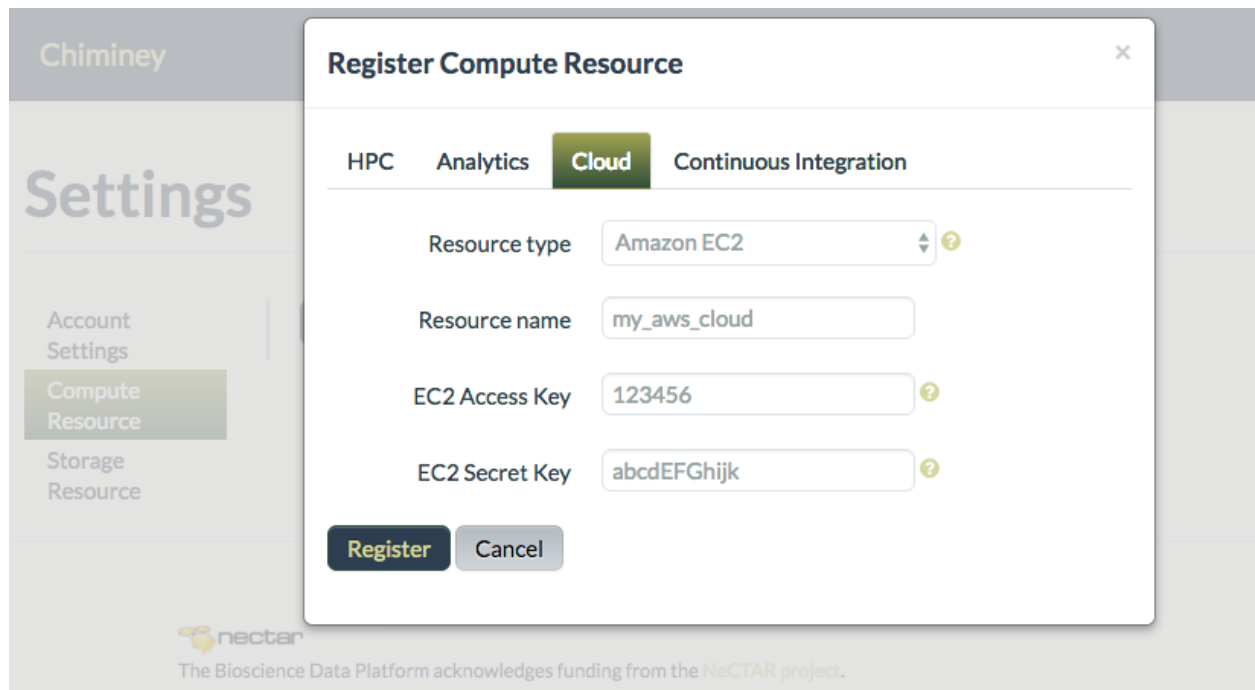
2.2.1 Registering Compute Resources

Various computing infrastructure and tools can be registered as compute resources. These resources are broadly categorised under *cloud*, *high performance computing (HPC)*, *analytics*, and continuous integration resources.

Cloud Compute Resource

1. Navigate to the Chiminey portal.

2. Log in with your credentials.

3. Click `Settings`.

4. Click `Compute Resource` from the `Settings` menu.

5. Click `Register Compute Resource`

6. Click the `Cloud` tab.

7. Select the resource type from the drop down menu. You may have access to more than one type of cloud service, e.g., NeCTAR and Amazon.

8. Enter a unique resource name.

9. Enter credentials such as EC2 access key and EC2 secret key

10. Click `Register`. The newly registered cloud-based compute resource will be displayed under *Cloud - NeCTAR/CSRack/Amazon EC2*.

Fig. 2.1: Figure. Registering a cloud-based compute resource

HPC Compute Resource

1. Navigate to the Chiminey portal.

2. Log in with your credentials.

3. Click `Settings`.

4. Click `Compute Resource` from the `Settings` menu.

5. Click `Register Compute Resource`

6. Click the `HPC` tab.

7. Enter a unique resource name.

8. Enter IP address or hostname of the HPC cluster head node or the standalone server.

9. Enter credentials, i.e. username and password. Password is not stored in the Chiminey platform. It is temporarily kept in memory to establish a private/public key authentication from the Chiminey platform to the resource.

10. Click `Register`. The newly registered resource will be displayed under *HPC - Cluster or Standalone Server* list.

Fig. 2.2: Figure. Registering a HPC compute resource

Analytics Compute Resource

1. Navigate to the Chiminey portal.

2. Log in with your credentials.

3. Click `Settings`.

4. Click `Compute Resource` from the `Settings` menu.

5. Click `Register Compute Resource`

6. Click the `Analytics` tab.

7. Select `Hadoop MapReduce` as the resource type from the drop down menu.

8. Enter a unique resource name.

9. Enter IP address or hostname of the Hadoop MapReduce resource.

10. Enter username and password. Password is not stored in the Chiminey platform. It is temporarily kept in memory to establish a private/public key authentication from the Chiminey platform to the resource.

11. Click `Register`. The newly registered resource will be displayed under *Analytics - Hadoop MapReduce* list.

Fig. 2.3: Figure. Registering an analytics compute resource (Hadoop MapReduce)

2.2.2 Registering Storage Resources

Remote file systems and data curation services like Mytardis mytardis_storage are used as a storage resources.

Remote File System

1. Navigate to the Chiminey portal.

2. Log in with your credentials.

3. Click `Settings`.

4. Click `Storage Resource` from the `Settings` menu.

5. Click `Register Storage Resource`

6. Click the `Remote File System` tab.

7. Enter a unique resource name.

8. Enter IP address or hostname of the remote file system.

9. Enter credentials, i.e. username and password. Password is not stored in the Chiminey platform. It is temporarily kept in memory to establish a private/public key authentication from the Chiminey platform to the resource.

10. Click `Register`. The newly registered resource will be displayed under *Remote File System* list.

Fig. 2.4: Figure. Registering a remote file system as a storage resource

MyTardis Storage Resource

1. Navigate to the Chiminey portal.

2. Log in with credentials

3. Click `Settings`

4. Click `Storage Resource` from the `Settings` menu

5. Click `Register Storage Resource`

6. Click the `MyTardis` tab.

7. Enter a unique resource name.

8. Enter IP address or hostname of the MyTardis instance.

9. Enter credentials, i.e. username and password. Username and password are stored on the Chiminey platform.

10. Click `Register`. The newly registered storage resource will be displayed under *MyTardis* list.
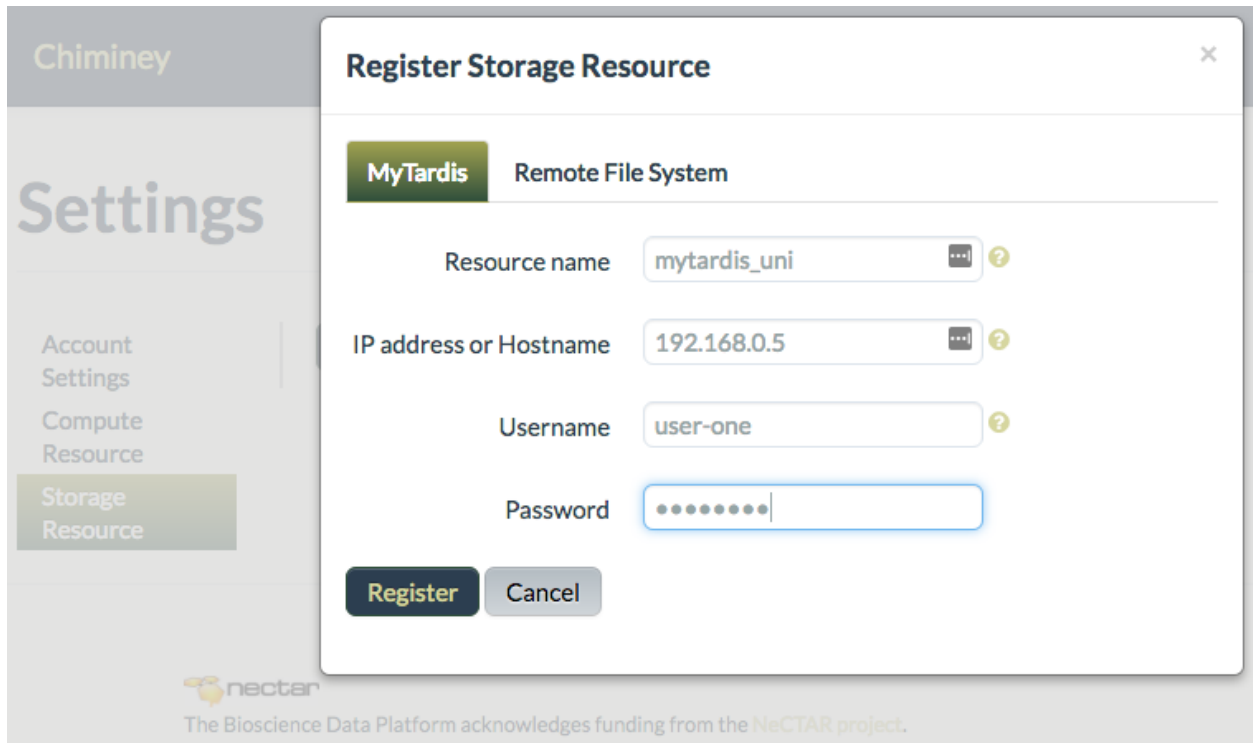
Fig. 2.5: Figure. Registering MyTardis, a data curation service.

2.2.3 Updating Resources

Follow the steps below to change he details of registered resources.

1. Navigate to the Chiminey portal.

2. Log in with credentials

3. Click `Settings`

4. From the `Settings` menu, depending on which resource you wish to update, click either *Compute Resource* or *Storage Resource*. All registered resources will be listed.

5. Locate the resource you wish to update, then click *Update*.

6. Make the changes, and when finished click `Update`

2.2.4 Removing Resources

In order to remove a registered resource, follow all the steps from *Updating Resources* but click *Remove* instead of *Update*. The resource will be removed from the resources' list.

2.3 Job Management

The end-user *submits*, *monitors*, and *terminates* jobs via the Chiminey portal. Before going into details about job management, we first discuss the *Chiminey UI* for submitting jobs.

2.3.1 The Job Submission UI

Fig. 2.6: Figure. Job Submission UI for wordcount smart connector

The aim of the following discussion is to understand the job submission UI, and therefore be able to run any smart connector job without difficulty. The job submission UI is accessed by clicking `Create Job` tab. The above figure shows the job submission UI of *wordcount* smart connector. The Chiminey job submission UI is composed of a list of

activated smart connectors, and the submission form of the selected smart connector. The submission form is divided into various sections. In general, each submission form has at least three sections: *1. Presets*, *2. Compute Resource* and *3. Locations*.

1. Presets

The end-user can save the set of parameters values of a job as a preset. Each preset must have a unique name. Using the unique preset name, the end-user can retrieve, update and delete saved presets.

2. Compute Resource

This section includes the parameters that are needed to utilise the compute resource associated with the given smart connector. For instance, hadoop compute resources need only the name of the registered hadoop cluster (see *Analytics Compute Resource*), while the cloud compute resource needs the resource name as well as the total VMs that can be used for the computation. Note that the names of all registered compute resources are automatically populated to a dropdown menu on the submission form.

3. Locations

These parameters are used to specify either input or output directories on a registered storage resource. Each location consists of two parameters: a storage location and a relative path. `Storage location` is a drop-down menu that lists the name of all registered storages and, their corresponding root path. A *root path* is an absolute path to the directory on the storage resource onto which all input and output files will be saved. `Relative path` is the name of a subdirectory of the root path that contains input and/or output files. In the case of input locations, Chiminey retrieves the input files that are needed to run the smart connector job from this subdirectory. In the case of output location, Chiminey will save the output of the smart connector job to the subdirectory.

4. Optional Parameter Sections

Some job submission forms include one or more of the following sections:

Reliability

> Fault tolerance support is provided to each smart connector job. How- ever, the enduser can limit the degree of such support using the reliability parameters: reschedule failed processes and maximum retries.

Sweep

> Sweep allows end-users to run multiple jobs simultaneously from a single submission. The sweep allows end-users to provide ranges of input values for parameters, and the resulting set of jobs produced span all possible values within that parameter space. These ranges of parameters are defined at job submission time, rather than being hard-coded in the definition of the smart connector. The common use- cases for this feature are to generate multiple results across one or more variation ranges for later comparison, and to quickly perform experi- mental or ad-hoc variations on existing connectors. Endusers specify the parameter(s) and their possible values via the sweep parameter.

Data curation resource

This section provides the parameters that are needed to curate the output of a smart connector job. The section includes a drop- down menu that is populated with the name registered data curation services like MyTardis.

Domain-specific parameters

These parameters are needed to guide the execution of the domain-specific payload of a given smart connector. wordcount smart connector has *Word Pattern* while hrmclite has *pottype*, *error threshold*, and others.

2.3.2 Job Submission

Follow the steps below

1. Navigate to the Chiminey portal

2. Log in with your credentials

3. Click `Create Job` from the menu bar

4. Select the smart connector from the list of smart connectors

5. Enter the values for the parameters of the selected smart connector.

6. Click `Submit Job` button, then `OK`

2.3.3 Job Monitoring

Once a job is submitted, the end-user can monitor the status of the job by clicking `Jobs tab`. A job status summary of all jobs will be displayed. The most recently submitted job is displayed at the top. Click `Info` button next to each job to view a detailed status report.

2.3.4 Job Termination

The *Jobs* page also allows to terminate submitted jobs. To terminate a job, check the box at the end of the status summary of the job, click `Terminate selected jobs` button at the end of the page. The termination of the selected jobs will be scheduled. Depending on the current activity of each job, terminating one job may take longer than the other.

2.4 Presets Management

The end-user can save the set of parameters values of a job as a preset. Each preset must have a unique name. Using the unique preset name, the end-user can retrieve, update and delete saved presets.

2.4.1 Adding Preset

1. Fill the parameter values for the job you are about to submit

2. Click **+** button next to the `Preset Name` drop down menu

Fig. 2.7: Figure. Monitoring a job

Fig. 2.8: Figure. Terminating a job
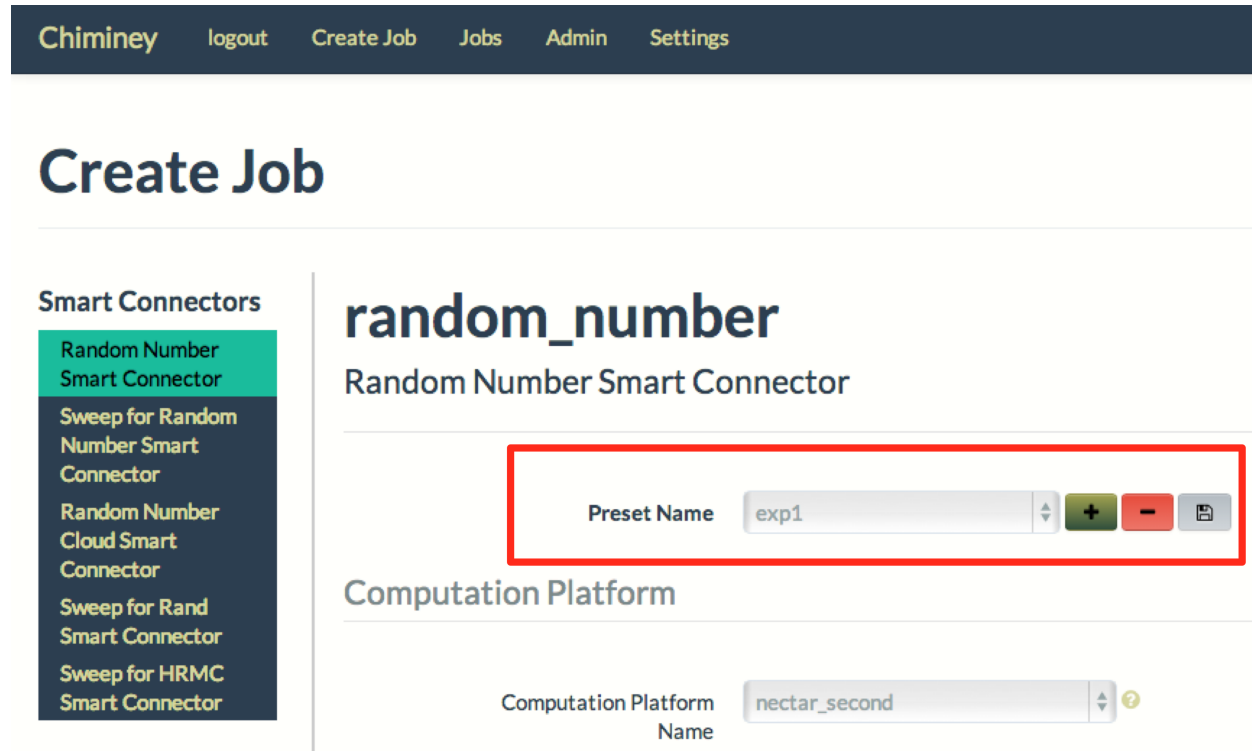
Fig. 2.9: Figure. Managing presets

3. Enter a unique name for the new preset

4. Click `Add`

2.4.2 Retrieving Preset

1. Select the preset name from the `Preset Name` drop down menu. The parameters on the submit job will be filled using parameters values that are retrieved from the selected preset.

2.4.3 Updating Preset

1. Select the preset name from the 'Preset Name' drop down menu.

2. Change the value of parameters as needed

3. Save your changes by clicking the save button next to the `Preset Name` drop down menu.

2.4.4 Deleting Preset

1. Select the preset name from the `Preset Name` drop down menu.

2. Click **-** button next to the 'Preset Name' drop down menu. Then, confirmation box appears.

3. Click `OK` to confirm.

Developer Manual

3.1 Smart Connector: the core concept within Chiminey

A **smart connector** is the core concept within Chiminey that enables endusers to perform complex computations on distributed computing facilities with minimal effort. It uses the abstractions provided by Chiminey to define transparent automation and error handling of complex computations on the cloud and traditional HPC infrastructure.

3.1.1 Stage

A **stage** is a unit of computation within Chiminey. Each stage hast at least the following elements:

- **validation:** Before the execution of a smart connector starts, the Chiminey server checks whether the constraints of all stages of the smart connector are met. This is done by invoking `input_valid(self, ...)` method of each stage of the smart connector.

- **pre-condition:** The Chiminey platform uses pre-conditions to determine the stage that should be executed next. The Chiminey platform invokes the method `is_triggerred(self, ...)` in order to check whether the pre-condition of a particular stage is met.

- **action:** This is the main functionality of a stage. Such functionality includes creating virtual machines, waiting for computations to complete, and the like. Once the Chiminey platform determines the next stage to execute, the server executes the stage via the `process(self, ...)` method.

- **post-condition:** This is where the new state of the smart connector job is written to a persistent storage upon the successful completion of a stage execution. During the execution of a stage, the state of a smart connector job changes. This change is saved via the `output(self, ...)` method.

3.1.2 The relationship between smart connectors and stages

A smart connector is composed of stages, each stage with a unique functionality. Following are the predefined stages that make up a smart connector (the predefined stages are located at `chiminey/corestages`):

- **parent:** Provides a handle to which all stages are within a smart connector are attached when a smart connector is registered within Chiminey. Contains methods that are needed by two or more stages.

- **configure:** Prepares scratch spaces, creates MyTardis experiments, ...

- **create:** Creates virtual machines on cloud-based infrastructure.

- **bootstrap:** Sets up the execution environment for the entire job, e.g. installs dependencies.

- **schedule:** Sets up the execution environment for individual task, and schedules tasks to available resources. A job is composed of one or more tasks. This stage is especially important when the job has more than one task.

- **execute:** Starts the execution of each task.

- **wait:** Checks whether a task is completed or not. Collects the output of completed tasks.

- **transform:** Prepares the input to the computation in the next iteration. Some smart connector jobs, for example Hybrid Reverse Monte Carlo simulations, have more than one iterations. When all tasks in the current iteration are completed and their corresponding output is collected, the transform stage prepares the input to the upcoming tasks in the next iteration.

- **converge:** Checks whether convergence is reached, where a job has more than one iteration. A convergence is assumed to be reached when either some criterion or the maximum number of iterations is reached.

- **destroy:** Terminates previously created virtual machines.

3.1.3 Creating a smart connector

Creating a smart connector involves completing three tasks:

1. providing *the core functionality* of the smart connector,

2. attaching *resources and optional non-functional properties*, and

3. *registering* the new smart connector with the Chiminey platform.

Each of the three tasks is discussed below by creating an example smart connector. This smart connector generates a random number with a timestamp, and then writes the output to a file.

NB: Login to the Chiminey docker container.

- For Mac OS X and Windows users, open *Docker Quickstart Terminal*. For linux-based OS users, login to your machine and open a terminal.

- Login to the chiminey docker container:

```
$ cd docker-chiminey
$ ./chimineyterm
```

The Core Function

The core functionality of a smart connector is provided either via a *payload* or by overriding the `run_task` method of `chiminey.corestages.execute.Execute` class. In this example, we use a minimal payload to provide the core functionality of this smart connector. Thus, we will prepare the following payload.

```
payload_randnum/
|--- process_payload
|    |--- main.sh
```

Below is the content of main.sh:

```
#!/bin/sh
OUTPUT_DIR=$1
echo $RANDOM > $OUTPUT_DIR/signed_randnum date > $OUTPUT_DIR/signed_randnum
# --- EOF ---
```

Notice `OUTPUT_DIR`. This is the path to the output directory, and thus Chiminey expects all outputs to be redirected to that location. The contents of `OUTPUT_DIR` will be transferred to the output location at the end of each computation.

Attaching resources and non-functional properties

Resources and non-functional properties are attached to a smart connector by overriding `get_ui_schema_namespace` method of `chiminey.initialisation.coreinitial.CoreInitial` class. New domain-specific variables can be introduced via `get_domain_specific_schemas` method. In this example, we will need to attached a unix compute resource for the computation, and a storage resource for the output location. However, we will not add a non-functional property.

Under chiminey/, we create a python package *randnum*, and add `initialise.py` with the following content

```
from chiminey.initialisation import CoreInitial
from django.conf import settings
class RandNumInitial(CoreInitial):
def get_ui_schema_namespace(self):
        schemas = [
                settings.INPUT_FIELDS['unix'],
                settings.INPUT_FIELDS['output_location'],
] return schemas
# ---EOF ---
```

NB: The list of available resources and non-functional properties is given by `INPUT_FIELDS` parameter in `chiminey/settings_changeme.py`

Registration

The final step is registering the smart connector with the Chiminey platform. The details of this smart connector will be added to the dictionary `SMART CONNECTORS` in `chiminey/settings changeme.py`. The details include a unique name (with no spaces), a python path to `RandNumInitial` class, the description of the smart connector, and the absolute path to the payload.

```
"randnum": {
        "name": "randnum",
        "init": "chiminey.randnum.initialise.RandNumInitial",
        "description": "Randnum generator, with timestamp",
        "payload": "/opt/chiminey/current/payload_randnum"
},
```

Finally, restart the Chiminey platform and then activate `randnum` smart connector. You need to exit the docker container in order to restart:

```
$ exit
$ sh restart
$ ./activatesc randnum
```

Food for Thought

In the example above, we created a smart connector that generates a random number on a unix-based machines. Even though the random number generator a simple smart connector, the tasks that are involved in creating a smart connector for complex programs is similar. If your program can be executed on a cloud, HPC cluster, hadoop cluster, then this program can be packaged as a smart connector. The huge benefit of using the Chiminey platform to run your program is you don't need to worry about how to manage the execution of your program on any of the provided compute resources. You can run your program on different types of compute resources with minimal effort. For instance, to generate random on a cloud-based virtual machine, we need to change only one word in `get_ui_schema_namespace` method. Replace `unix` by `cloud`. Then, restart Chiminey, and activate your cloud-based random number generator.

Check the *various examples* are given in this documentation. These examples discuss the different types of compute and storage resources, and non-functional properties like reliability and parameter sweep.

3.2 Parameter Sweep

The Chiminey platform provides two types of parameter sweeps:

- *External parameter sweep*
- *Internal parameter sweep*

3.2.1 External Parameter Sweep

External parameter sweep allows end-users to simultaneously submit and run multiple jobs. The external sweep gives power to the end-users to provide a range of input values for a set of parameters of their choice, and the resulting set of jobs span all possible values from that parameter space.

The set of parameters are defined as part of input data preparation, rather than being "hard-coded" to the definition of the smart connector. This is done using templating language. Here are the steps:

1. Identify the input files that contain the parameters of your choice.

2. Surround each parameter name with double curly brackets. Suppose an input file entry is `var1 15`. Replace this line by `{{var1}} 15`; where `15` is the default value.

3. Save each input file as `filename_template`.

The end-user provides the range of values of these parameters via a *sweep map* during job submission.

The common usecases for the external parameter sweep are to generate multiple results across one or more variation ranges for later comparison, and to quickly perform experimental or ad-hoc variations on existing smart connectors.

3.2.2 Internal Parameter Sweep

Internal parameter sweep allows developers to create a smart connector that spawns multiple independent tasks during the smart connector's job execution. When a smart connector is created, the developer includes a set of parameters that will determine the number of tasks spawned during the execution of the smart connector.

The developer uses a *sweep map* to specify a range of values for the selected set of parameters during a smart connector definition. This is done by subclassing `Parent`, which is located at `chiminey/corestages/parent.py`, and overwriting the method `get_internal_sweep_map(self, ...)` to include the new sweep map. The default sweep map generates one task.

```python
def get_internal_sweep_map(self, settings, **kwargs):
    rand_index = 42
    map = {'val': [1]}
    return map, rand_index
```

NB: A smart connector job that is composed of multiple tasks, due to an internal parameter sweep, is considered to be complete when each task either finishes successfully or fails beyond recovery.

3.2.3 Sweep Map

A sweep map is a JSON dictionary that is used to specify a range of values for a set of parameters.

- The key of the dictionary corresponds to a parameter name.

- The value of the dictionary is a list of possible values for that parameter.

Below is an example of a sweep map.

```
sweep_map = {"var1": [7,9], "var2": [42]}
```

The cross-product of the values of the parameters in a sweep map is used to determine the minimum *[0] number of tasks or jobs generated during job submission. The above sweep map, for example, generates

- **two jobs** if used for external parameter sweep, or

- **two tasks** per job if used for internal parameter sweep

Imapct of unknown parameters in a sweep map

An **unknown parameter** is a parameter that is not needed during the execution of a smart connector. A **known parameter**, on the other hand, is a parameter whose value is needed for the correct functioning of a smart connector.

Including an unknown parameter in a sweep map does not have any ill-effect during execution. However, this parameter causes an increase in the number of generated tasks/jobs, provided that the number of the values of the unknown parameter is more than one.

Suppose `var1` and `var2` are known parameters, and `x` is an unknown parameter of a specific smart connector; and the sweep map is `{"var1": [7,9], "var2": [42], "x": [1,2]}`.

- If the sweep map is used for external parameter sweep, the number of jobs doubles due to the inclusion of parameter `x` with two values. If the sweep map is used for internal sweep, the number of tasks doubles as well. If the value of `x` is changed to `[1,2,3]`, the number of jobs/tasks triples, and so on.

The additional jobs or tasks waste computing, storage and network resources. However, there are cases where such feature is useful.

- The end-user can use this feature to run jobs with identical inputs, and then compare whether the jobs produce the same output.

- If each task has unpredictable output irrespective of other variables being constant, the developer can use the feature to run many of these tasks per job, each task with different output. For example, generating a random number without fixing the seed almost always guarantees a new number.

See also:

External parameter sweep:

- *Unix-based smart connector with external parameter sweep*

Internal parameter sweep:

- Cloud-based smart connector with internal parameter sweep

3.3 Payload

A **payload** is a set of system and optionally domain-specific files that are needed for the correct execution of a smart connector. The *system files* are composed of bash scripts while the *domain-specific files* are developer provided executables. The system files enable the Chiminey platform to setup the execution environment, execute domain-specific programs, and monitor the progress of setup and execution.

---

[0] The total number of tasks that are generated per job depends on the type of the smart connector. In addition to the sweep map, domain-specific variables or constraints play a role in determining the number of tasks per job.

NB: All smart connectors that are executed on a cloud and a cluster infrastructure must have a payload. However, smart connectors that are executed on unix servers do not need a payload unless the execution is asynchronous.

Below is the structure of a payload.

```
payload_name/
|--- bootstrap.sh
|--- process_payload
|    |--- main.sh
|    |--- schedule.sh
|    |--- domain-specific executable
```

The names of the files and the directories under `payload_name`, except the domain specific ones, cannot be changed.

- `bootstrap.sh` includes instructions to install packages, which are needed by the smart connector job, on the compute resource.

- `schedule.sh` is needed to add process-specific configurations. Some smart connectors spawn multiple processes to complete a single job. If each process needs to be configured differently, the instruction on how to configure each process should be recorded in schedule.sh.

- `main.sh` runs the core functionality of the smart connector, and writes the output to a file. Chiminey sends the path to input and output directories via command-line arguments:

   the first argument for input and the second for output. The smart connector developer can read inputs from the input directory, and redirect outputs of the job to the output directory. Upon the completion of the smart connector job, Chiminey will transfer the content of the output directory to the end-user's designated location.

- `domain-specific executables` are additional files that may be needed by main.sh.

Not all smart connector jobs require new packages to be installed, process-level configuration or additional domain-specific executables. On such cases, the minimal payload, as shown below, can be used.

```
payload_name/
|--- process_payload
|    |--- main.sh
```

NB: Sample payloads are available under each example smart connector, at the Chiminey Github Repository.

3.4 Chiminey User Interface

The Chiminey platform automatically generates a job submission web page for each smart connector. However, this web page contains only a drop down menu of *presets*. The web page will also contain a *parameter sweep* input field for smart connectors with a sweep feature. Since these two input fields are not sufficient to submit a job, the developer should specify the input fields that are needed to submit a particular smart connector job. This is done during the *definition of the smart connector*.

Within the Chiminey platform, there are various *input field types*, organised in groups like compute resource variables, location variables and domain-specific variables. The input fields, except the domain-specific ones, are provided via `INPUT_FIELDS` parameter in `chiminey/settings_changeme.py`. The following table shows the list of input field types and their description.

Input Field Type	Description
`unix`	Dropdown menu containing the registered HPC compute resources.
`cloud`	Dropdown menu of registered cloud resources, number of VMs to be used for the job.
`hadoop`	Dropdown menu of registered hadoop clusters.
`output_location`	Dropdown menu of registered storage resources (i.e. remote file system) with root path, and a text field for specifying directories under the root path.
`input_location`	Same as output location.
`location`	Input and output location.
`reliability`	Set of fields to control the degree of the provided fault tolerance support.
`mytardis`	Set of fields to enable end users to curate the input and output of their smart connector job on MyTardis.
`hrmclite`	Domain-specific input fields needed to run *HRMCLite* jobs.
`hrmc`	Domain-specific input fields needed to run *HRMC* jobs.
`wordcount`	Domain-specific input fields needed to run *wordcount* jobs.

3.4.1 Constructing Smart Connector Input Fields

Here, we see how to include the input fields that are needed for submitting a smart connector job. *When a smart connector is created*, one of the tasks is attaching resources and non-functional properties via input field types. This task is done by overriding `get_ui_schema_namespace(self)` of the `CoreInitial` class. The `CoreInitial` class is available at `chiminey/initialisation/coreinitial`.

Suppose the new smart connector is cloud-based and writes its output to a unix server. Therefore, the job submission page of this smart connector must include two input field types that enables end-users to provide a) a cloud-based compute resource and b) an output location. Suppose `CloudSCInitial` extends the `CoreInitial` class:

```
from chiminey.initialisation import CoreInitial
from django.conf import settings
class CloudSCInitial(CoreInitial):
def get_ui_schema_namespace(self):
    schemas = [
            settings.INPUT_FIELDS['cloud'],
            settings.INPUT_FIELDS['output_location'],
] return schemas


# ---EOF ---
```

Including domain-specific input fields

Input field types that are included within the Chiminey platform are generic and are included within the platform. However domain-specific input fields must be defined when needed. A domain-specific input field type is provided by overriding `get_domain_specific_schemas(self)` of the `CoreInitial` class. This method will return a list of two elements:

1. The description of the input field type e.g. *HRMCLite Smart Connector*

2. A dictionary whose keys are the names of domain-specific input fields, their values are dictionaries with the following keys:

   - **type**: There are three types of input fields: *numeric* (models.ParameterName.NUMERIC), *string* (models.ParameterName.STRING), *list of strings* (models.ParameterName.STRLIST). *numeric* and *string* inputs have a text field while a *list of strings* has a drop-down menu. Enduser inputs are validated against the type of the input field.

   - **subtype**: Subtypes are used for additional validations: *numeric* fields can be validated for containing whole and natural numbers.

   - **description**: The label of the input field.

   - **choices**: If the type is *list of strings*, the values of the dropdown menu is provided via *choices*.

   - **ranking**: Ranking sets the ordering of input fields when the fields are displayed.

   - **initial**: The default value of the field.

   - **help_text**: The text displayed when a mouse hovers over the question mark next to the field.

Below are two examples of domain-specific input field types: *wordcount* and *HRMCLite* smart connector.

   - WordCount smart connector input field type

```
def get_domain_specific_schemas(self):
        schema_data =  [u'Word Count Smart Connector',
            {
                u'word_pattern': {'type': models.ParameterName.STRING,
                                  'subtype': 'string',
```

```
                                        'description': 'Word Pattern',
                                        'ranking': 0,
                                        'initial': "'[a-z.]+'",
                                        'help_text': 'Regular expression of filtered words'},
            }
        ]
    return schema_data
```

- HRMCLite smart connector input field type

```
def get_domain_specific_schemas(self):
    schema_data = [u'HRMCLite Smart Connector',
        {
            u'iseed': {'type': models.ParameterName.NUMERIC,
                       'subtype': 'natural',
                       'description': 'Random Number Seed',
                       'ranking': 0,
                       'initial': 42,
                       'help_text': 'Initial seed for random numbers'},
            u'pottype': {'type': models.ParameterName.NUMERIC,
                         'subtype': 'natural',
                         'description': 'Pottype',
                         'ranking': 10,
                         'help_text': '',
                         'initial': 1},
            u'error_threshold': {'type': models.ParameterName.STRING,
                                 'subtype': 'float',
                                 'description': 'Error Threshold',
                                 'ranking': 23,
                                 'initial': '0.03',
                                 'help_text': 'Delta for iteration convergence'},
            u'optimisation_scheme': {'type': models.ParameterName.STRLIST,
                                     'subtype': 'choicefield',
                                     'description': 'No. varying parameters',
                                     'ranking': 45,
                                     'choices': '[("MC","Monte Carlo"), ("MCSA", "Monte Carlo with S
                                     'initial': 'MC', 'help_text': '',
                                     'hidefield': 'http://rmit.edu.au/schemas/input/hrmc/fanout_per_
                                     'hidecondition': '== "MCSA"'},
            u'fanout_per_kept_result': {'type': models.ParameterName.NUMERIC,
                                        'subtype': 'natural',
                                        'description': 'No. fanout kept per result',
                                        'initial': 1,
                                        'ranking': 52,
                                        'help_text': ''},
            u'threshold': {'type': models.ParameterName.STRING,
                           'subtype': 'string',
                           'description': 'No. results kept per iteration',
                           'ranking': 60,
                           'initial': '[1]',
                           'help_text': 'Number of outputs to keep between iterations. eg. [2] would
            u'max_iteration': {'type': models.ParameterName.NUMERIC,
                               'subtype': 'whole',
                               'description': 'Maximum no. iterations',
                               'ranking': 72,
                               'initial': 10,
                               'help_text': 'Computation ends when either convergence or maximum ite
        }
```

```
      ]

   return schema_data
```

3.5 Examples

Here, we use the following examples to show the different features of a smart connector and how a smart connector is defined and registered within a Chiminey server.

3.5.1 Quick Example: Random Number Smart Connector

In this example, we create a basic smart connector that generates two random numbers on a unix machine, saves the numbers to a file, and then transfers the file to a provided output location. The unix machine must have ssh service enabled.

This smart connector has already been discussed in section *Creating a smart connector*.

3.5.2 Word Count Smart Connector

3.5.3 HRMCLite Smart Connector

3.5.4 The Hybrid Reverse Monte Carlo (HRMC) Smart Connector

Note: This example is significantly more complicated than the previous examples. Therefore we describe here the unique features of this connector and invite the reader to read the source code for this connector in detail. It combines a number of features from the previous examples and uses the same overall architecture.

The Hybrid Reverse Monte Carlo Smart Connector, hereafter HRMC SC, is designed to run *the implementation of an HRMC simulation by George Opletal*. The HRMC SC runs HRMC simulations on a cloud compute resource. It reads inputs from a remote file system, and then writes output to a remote file system *and* a data curation service, i.e. MyTardis. The HRMC SC enables end-users to control the degree of the provided fault tolerance support. Furthermore, this smart connector includes a sweep functionality to enable end-users to simultaneously execute multiple HRMC jobs from a single submission.

The HRMC SC and related topics will be discussed as follows:

- *HRMC source code*
- *The Core function*
- *Attaching resources and non-functional properties*
- *Registering the HRMC SC*

Hybrid Reverse Monte Carlo - Source Code Version 2.0 (Oct 2012)

Code development by:

Dr. George Opletal

g.opletal@gmail.com

Applied Physics RMIT, Melbourne Australia.

Contributions to code development:

Dr. Brendan O'Malley

Dr. Tim Petersen

Published in:

7. Opletal, T. C. Petersen, I. K. Snook, S. P. Russo, HRMC_2.0: Hybrid Reverse Monte Carlo method with silicon, carbon and germanium potentials, Com. Phys. Comm., 184(8), 1946-1957 (2013).

License: CPC License: http://cpc.cs.qub.ac.uk/licence/licence.html

HRMC Core Function

The core functionality of the HRMC SC is provided through a *payload*. The HRMC payload is similar to the following.

```
payload_hrmc/
|--- bootstrap.sh
|--- process_payload
|    |--- HRMC2.tar.gz
|    |--- PSDCOde.tar.gz
|    |--- schedule.sh
|    |--- main.sh
```

The HRMC SC requires packages like dos2unix, fortran compiler. Thus, all the required dependancies are specified in `bootstrap.sh`. The content of `bootstrap.sh` is as follows:

```
#!/bin/bash

 yum -y install dos2unix gcc-gfortran compat-libgfortran-41 gcc-gfortran.x86_64
```

The payload includes domain-specific executables, i.e. `HRMC2.tar.gz` and `PSDCOde.tar.gz`. The `schedule.sh` of HRMC SC contains process-specific configurations. `schedule.sh` is responsible to extract the executables for each HRMC process. Below shows the content of `schedule.sh`.

```
#!/bin/bash
# version 2.0

INPUT_DIR=$1
OUTPUT_DIR=$2

tar --extract --gunzip --verbose --file=HRMC2.tar.gz
f95 HRMC2/hrmc.f90 -fno-align-commons -o HRMC

tar --extract --gunzip --verbose --file=PSDCode.tar.gz
gfortran PSDCode/PSD.f -o PSDCode/PSD
```

`main.sh` is the core of HRMC SC. Recall that *Chiminey sends the path to input (''INPUT_DIR'*) and output (OUTPUT_DIR) directories via command-line arguments<payload>'. Here, the SC developer moves the HRMC executable, which was extracted by `schedule.sh`, to the input directory. The SC developer changes its working directory to `INPUT_DIR`, run HRMC, and redirect the execution output to `OUTPUT_DIR`. When HRMC is completed, the SC developer moves additional files to `OUTPUT_DIR`. As the next major step, the SC developer runs the `PSD` executable, which was extracted by `schedule.sh`. Once the execution is completed, the necessary files are moved to `OUTPUT_DIR`.

NB: Running HRMC and PSD takes a long time. However, the SC developer should not be concerned about this as Chiminey will ensure that all tasks are run asynchronously.

```
#!/bin/bash
# version 2.0

INPUT_DIR=$1
OUTPUT_DIR=$2

cp HRMC $INPUT_DIR/HRMC
cd $INPUT_DIR
./HRMC >& ../$OUTPUT_DIR/hrmc_output
cp input_bo.dat ../$OUTPUT_DIR/input_bo.dat
cp input_gr.dat ../$OUTPUT_DIR/input_gr.dat
cp input_sq.dat ../$OUTPUT_DIR/input_sq.dat
cp xyz_final.xyz  ../$OUTPUT_DIR/xyz_final.xyz
cp HRMC.inp_template ../$OUTPUT_DIR/HRMC.inp_template
cp  data_errors.dat   ../$OUTPUT_DIR/data_errors.dat

cp -f xyz_final.xyz ../PSDCode/xyz_final.xyz
cd ../PSDCode; ./PSD >&  ../$OUTPUT_DIR/psd_output
cp PSD_exp.dat ../$OUTPUT_DIR/
cp psd.dat ../$OUTPUT_DIR/
```

Attaching Resources and Non-functional properties

Registering the HRMC SC

Setup

As with the previous examples, we setup the new connector payload:

```
mkdir -p /var/chiminey/remotesys/my_payloads
cp -r  /opt/chiminey/current/chiminey/examples/hrmc2/payload_hrmc /var/chiminey/remotesys/my_payloads
```

Then register the new connector within chiminey:

```
sudo su bdphpc
cd /opt/chiminey/current
bin/django hrmc
Yes
```

This example is significantly more complicated than the previous random number examples. Therefore we describe here the unique features of this connector and invite the reader to read the source code for this connector in detail. It combines a number of features from the previous examples and uses the same overall architecture.

The Input Directory

As described earlier, each connector in Chiminey system can elect to specify a *payload* directory that is loaded to each VM for cloud execution. This payload is fixed for each type of connector.

However, if practice you would likely want to vary the behaviour of *different* runs of the same connector, to change the way a process is executed or perform exploratory analysis.

This is accomplished in Chiminey by the use of a special *input directory*. This is one of the most powerful features of a smart connector as it allows individual runs to be fully parameterised on the initial input environment and environment during execution.

Ideally the payload directory would contains source or code for the application, and the input directory would contain configuration or input files to that application.

The input directory is a remote filesystem location (like the output directory) defined and populated before execution, which contains files that will be loaded into the remote copy of the payload after it has been transferred to the cloud node, *for every run*. Furthermore the contents of input files in that directory can be varied at run time.

Any files within the input directory can be made into a template by adding the suffix `_template` to the filename. Then, this file is interpreted by the system as a Django template file.

Consider the application "foo' has its source code in a payload, but requires a "foo.inp"file containing configuraiton information. For example:

```
# foo input file
iseed 10
range1 45
range2 54
fudgefactor 12
```

is an example input for one run. To parameterise this file you rename it to `foo.inp_template` and replace the values that need to vary with equivalent template tags:

```
# foo input file
iseed {{iseed}}
range1 {{range1}}
range2 {{range2}}
fudgefactor 12
```

The actual values to be used are substituted at runtime by the system. THe values can come from the external sweep map, the internal sweep map, domain-specific values in the submission page, and constant values set within the input directory.

For example, the `iseed` value may be an input field on the submission page, the `range1` value may be predefined to be constant during all runs, and the `range2` has to go between the values `50--52`.

This parameterisation is performed using a `values` file, which is a special file at the top of the input directory. This JSON dictionary contains values to be instantiated into template files at run time. The values files included in the original input directory can contain constant values that would then apply generally to any connector using that input directory.

For this example, we the directory would include a file `values` containing:

```
{
    "range1": 45
}
```

Then initially, all runs of `foo` will include:

```
range1 45
```

in the `foo.inp` file

However, Chiminey also automatically populates the values directory with other key/value s representing the data typed into the job submission page form fields, the specific values from the sweep map for *that* run. All these values can be used in instantiation of the template files.

For this example, if at jobs submission time the user entered `iseed` as 10, and the sweep map values as `{"range2": [50, 51]}` then external sweep will produce multiple processes each with a values file across the range `range2`. For example:

```
{
    "iseed": 10
    "range1": 45,
    "range2": 50,

}
```

or:

```
{
    "iseed": 10
    "range1": 45,
    "range2": 51,
}
```

The `foo.inp_template` file is matched against the appropriate `values` file, to create the required input file. For example:

```
# foo input file
iseed 10
range1 45
range2 50
fudgefactor 12
```

or:

```
# foo input file
iseed 10
range1 45
range2 51
fudgefactor 12
```

Hence these are are the `foo.inp` files for each run.

The use case for such a connector:

1. Prepare a payload containing all source code and instructions to compile as before.

2. Prepare a remote input directory containing all the input files needed by the computation. If the contents of any of these files need to vary, then rename the files and add `{{name}}` directives to identify the points of variation. Names are:

   (a) keys from the input schemas within the submission page.

   (b) constant values for the whole computation.

3. Optionally add a `./values` file containing a JSON dictionary of mappings between variables and values. These entries are constant values for the whole computation.

4. During execution, Chiminey will load up values files and propagate them in input and output directories, will put values corresponding to all input values, individual values from the space of sweep variables. These variables will be substituted into the template to make an original input file suitable for the actual execution.

In the HRMC connector, the `HRMC.inp` file is templated to allows substitution of values from both the job submission page and from the sweep variable. See `input_hrmc/initial` directory and the inluded `HRMC.inp_template` and `values` files.

Complex Internal Sweeps

The `randnuminternalsweep` connector defined a simple map in the parent stage that maps an input into two variations based on a variable `var`. While that value was not used in that example, we can see that if a input directory was used then each of the two variations would get different values for the `var` variable in the `values` file and could be used in any input template file.

For the HRMC smart connector, the mapping is significantly more complicated. In the `get_internal_sweep_map` method of `hrmcparent.py`, the map is definedin stages using existing variables (in the `values` file), the values in the original form, plus new variables based on random numbers and on the current iteration of the calculation. Thus the number of processes and their starting variables can be specialised and context sensitive and then instantiated into template files for execution.

Use of Iterations

In the random numbers the standard behaviour was to execute stages sequentially from `Configure` through to `Teardown`. However, in the HRMC example, we support an run_setting variable `system/id` which allows a set of stages to be repeated multiple times and two new core stages, `Transform` and `Converge`. These stages are specialised in the HRMC example:

- After the results are generated during the execution phase, the `HRMCTransform` stages calculates a criterion value (the `compute_psd_criterion` method). The execution results are then prepared to become input for a next iteration (the `process_outputs` method)

- In the `HRMCConverge` stage, the new criterion value is then compared a previous iterations' value and if the difference is less than a threshold, then the smart connector execution is stopped. Otherwise, the value `system/id` is incremented and the triggering state for the execution phase is created which causes these stages to be re-executed. Finally, to handle the situation where the criterion will diverges or is converging too slowly, the `HRMCConverge` stage also halts the computation is the `system/id` variable exceeds a fixed number of iterations.

See the `hrmctransform.py` and `hrmcconverge.py` modules for more details.

Complex Mytardis Interactions

The HRMC example, expands on the MyTardis experiment created in the randonnumber example.

As before the `HRMCConverage` defines a curate_data method, and the `HRMCTransform` and `HRMCConverge` define a `curate_dataset` method. However, the later methods are significantly more complicated than the previous example.

The `mytardis/create_datadata` method takes a function for the dataset_name as before, which has a more complicated implementation. However, this example also uses the `dfile_extract_func` argument which is a dict from datafile names to functions. For all contained datafiles within the dataset, their names are matched to this dictionary, and when found, the associated function is executed with a file pointer to that files *contents*. The function then results the graph metadata required.

For example, `HRMCTransform` includes as an argument for `mytardis.create_dataset`:

```
dfile_extract_func= {'psd.dat': extract_psd_func,
'PSD_exp.dat': extract_psdexp_func,
'data_grfinal.dat': extract_grfinal_func,
'input_gr.dat': extract_inputgr_func}
```

Here for any datafile in the new dataset named *psd.dat* chiminey will run:

```python
def extract_psd_func(fp):
    res = []
    xs = []
    ys = []
    for i, line in enumerate(fp):
        columns = line.split()
        xs.append(float(columns[0]))
        ys.append(float(columns[1]))
    res = {"hrmcdfile/r1": xs, "hrmcdfile/g1": ys}
    return res
```

Here the function returns a directionry containing mappings to two lists of floats extracted from the datafile `psd.dat`. This value is then added as a metadata field attached to that datafile. For example:

```
graph_info   {}
name         hrmcdfile
value_dict   {"hrmcdfile/r1": [10000.0, 20000.0, 30000.0, 40000.0, 50000.0, 60000.0, 70000.0, 80000.0
value_keys   []
```

This can then be data to be used by the dataset level graph `hrmcdset` described in the `dataset_paramset` argument of the `create_dataset` method.

API Reference

4.1 Chiminey Stage APIs

4.1.1 `mytardis` – MyTardis APIS

The MyTardis module provides functions that allow publishing Chiminey results to a connected MyTardis System, allowing the online storing, access and sharing capabilities of data and metadata.

Datastructures

paramset

Module Functions and Constants

Example:

```
def _get_exp_name_for_make(settings, url, path):
                return str(os.sep.join(path.split(os.sep)[-2:-1]))

            def _get_dataset_name_for_make(settings, url, path):
                return str(os.sep.join(path.split(os.sep)[-1:]))

            self.experiment_id = mytardis.create_dataset(
                settings=mytardis_settings,
                source_url=encoded_d_url,
                exp_id=self.experiment_id,
                exp_name=_get_exp_name_for_make,
                dataset_name=_get_dataset_name_for_make,
                experiment_paramset=[],
                dataset_paramset=[
                    mytardis.create_paramset("remotemake/output", [])]
                )
```

4.1.2 `storage` – Storage APIS

This package provides a file-like api for manipulating local and remote files and functions at locations specified by platform instances.

Datastructures

Module Functions and Constants

4.1.3 `sshconnection` – Manipulation of Remote Resources

Datastructures

ssh_client

Module Functions and Constants

4.1.4 `compute` – Execution Of Remote Commands

Using an open ssh_client connector from sshconnector, these commands execute commands remotely on the target server.

Datastructures

Module Functions and Constants

4.1.5 `messages` – Logging communication for Chiminey

This package, modelled off the django logging module, posts status messages for the BDP system. There are two classes of context for the user of this API:

1. Within stage implementation, messages will be displayed within the status field in the job list UI.
2. During job submission, messages will be displayed on the redirected page as a header banner.

Messages are processed by a separate high-priority queue in the celery configuration. Note that message delivery may be delayed due to celery priority or db exclusion on the appropriate context, so this function should not be used for real-time or urgent messages.

Datastructures

By convention, error messages are final messages in job execution to indicate fatal error (though job might be restarted via admin tool) and success is used to describe final successful execution of the job.

Module Functions and Constants

Send a msg at the required priority level, as per the django logging module.

Uses contextid field of settings to determine which running context to assign messages.

4.1.6 `run_settings` – Contextual Namespace for Chiminey

The current state of any BDP calculation is represented as a context model instance, with an associated run_settings instance. This run_setting serves as input, output and scratch symbol table for all job execution.

The contextual namespace is used for numerous purposes in the BDP, including:

- Input parameters for Directive submission UI dialog

- Single source of truth for building settings dicts for BDP API modules.

- Job execution state

- Diagnostics and visualisation of job progress

- Stage triggering and scheduling during directive execution

Conceptually run_settings is a set of parameter sets each of which is conformant to a predefined schemas, that are defined in the admin tool.

run_settings is a two-level dictionary, internally serialised to models as needed.

Datastructures

context

A two level dictionary made up of schema keys and values. Conceptually, this structure is equivalent to a two-level python dictionary, but should be accessed via the API below. For example,

{ http://acme.edu.au/schemas/stages : { "key": id, "id": 3} }

Keys are concatenation of schema namespace and name, for example: http://acme.edu.au/schemas/stages/key

Module Functions and Constants

`chiminey.runsettings.`**`getval`**(*context*, *key*)
Retrieves the current value of the key within the run setttings context.

> **Parameters**
>
> > - **context** – current context
> >
> > - **key** – the key to search
>
> **Raises** SettingNotFoundException if the schema part of the key is not in the context
>
> **Returns** the value for the key

`chiminey.runsettings.`**`setval`**(*context*, *key*, *value*)
Sets the value of the key within the run settings context.

> **Parameters**
>
> > - **context** – current context
> >
> > - **key** – the key to search
>
> **Throws** runsettings.SettingNotFoundException if the schema part of key is not in the context.
>
> **Throws** runsettings.IncompatibleTypeException if the value is not compatible with the type of the key, according to the associated schema definition.

4.1.7 `cloudconnection` – Cloud Connection

Datastructures

Module Functions and Constants

4.1.8 `corestages` – Processing Steps in a directive

This is an abstract class that forms the interface for all directives, both smart connectors and utilties to provides steps in a calcuation.

Datastructures

Module Functions and Constants

4.2 `simpleui` – UI view members

Indices and tables

- genindex
- modindex
- search

G

getval() (in module chiminey.runsettings), 39

S

setval() (in module chiminey.runsettings), 39