Platform Migrator

Munir Contractor, Christophe Pradal, Dennis Shasha

Platform Migrator

Technical Report TR2018-990

Munir Contractor
mmc691@nyu.edu

Christophe Pradal
christophe.pradal@inria.fr

Dennis Shasha
shasha@cs.nyu.edu

May 12, 2018

CONTENTS:

CHAPTER

ONE

ABSTRACT

Currently, one of the major problems in software development and maintenance, specially in academia, is managing packages across time and systems. An application developed under a particular package manager using a certain set of packages does not always work reliably when ported to a different system or when abandoned for a period of time and picked up again with newer versions of the packages. In this report, we provide and describe Platform Migrator, a software that makes it easy to test applications across systems by identifying various packages in the base system, figuring out their corresponding equivalents in the new system and testing whether the software works as expected on the new system. Platform migrator can migrate software written and set up inside a conda environment to any Linux based system with conda or some other package manager. The philosophy of platform migrator is to identify a closure of the required dependencies for the software being migrated using the conda environment metadata and then use that closure to install the various dependencies on the target system. This documentation provides comprehensive details on how to use platform migrator and what it does internally to migrate software from one system to another. It also contains tutorials and case studies that can be replicated for better understanding of the process.

CHAPTER

TWO

PLATFORM MIGRATOR INTRODUCTION

Platform migrator is a tool which helps with migrating software from system to another. Its main aim is to ease the process of installing dependencies required for the software. It interfaces with various package managers to install the required dependencies and runs tests to identify whether the installation was successful or not.

For platform migrator, a system is defined by a set of package managers provided as part of a configuration file. This set may include some or all of the package managers available on the machine. In case only some package managers are used, the system is defined only as the set of those package managers. This can be used to migrate dependencies of a software from one package manager to another, on the same machine.

The two systems could also be different OSes or running in a VM. As long as there is a package manager available, platform migrator can try to port software across the systems. The rest of the documentation uses the terms base system and target system to refer to these systems:

base system This is the system on which the software is currently installed within a conda environment. The current version of platform migrator requires that there is a conda installation which can provide the list of packages used by the software. The software itself also must be available in an executable form, either as a compiled object or as source code.

target system This is the system to which the software will be migrated. Platform migrator should be installed on this system. Conda is not required on this system, but there must be some package manager present that the user can use to install software.

The base system and the target system must be able to connect with each other over a network through HTTP or the user of the base system must also install platform migrator and email the zip file produced over to the target system.

2.1 Installation

Platform migrator requires Python >= 3.5 to be available on the target system. It can be installed directly from pip on the target system

```
pip install --user platform-migrator
```

Or, if you want to install a development version

```
pip install --user git+https://gitlab.com/mmc691/platform-migrator.git
```

Note: On some systems, you may need to specify `pip3` instead of `pip` to make sure that Python 3 is used. Run `pip --version` to check which version of Python the default pip installation uses.

If pip is not available, it can be installed by cloning the git repository or downloading a zip file from Gitlab repository and executing

```
git clone https://gitlab.com/mmc691/platform-migrator
cd platform-migrator
python3 setup.py install --user
```

2.2 Execution

This provides a *basic* overview of how platform migrator works. See the tutorials and the internals documentation for the full details on how to use platform migrator.

The whole process is executed in 4 main steps:

1. Start the server on the target system with `platform-migrator server start`. This starts an HTTP server which by default listens on localhost:9001.
2. On the base system, execute the following on the command line:

   ```
   curl http://<server-name>:<server-port>/migrate > script.py
   python script.py
   ```

   The server name and port are the hostname and port on which the server on the target system is listening. See *`base_sys_script`* documentation for details on what the script does.
3. Generate the test configuration file on the target system. See the *Writing test configuration files* section on how to do this.
4. Run `platform-migrator migrate <name> <config>` on the target system. See the *Tutorials* and *Case Studies* section for detailed description about this step.

Alternately, if a network connection cannot be setup between the systems, the software can be transported over email as a zip file. In this case, platform migrator must be installed on the base system as well as the target system. The steps will be:

1. Package the software on the base system using `platform-migrator pack` and send the zip file created by platform migrator to the target system.
2. Unpack the zip file on the target system using `platform-migrator unpack <zip-file>`.
3. Generate the test configuration file on the target system. See the *Writing test configuration files* section on how to do this.
4. Run `platform-migrator migrate <name> <config>` on the target system. See the *Tutorials* and *Case Studies* section for detailed description about this step.

If the migration was successful, the software will be saved in the configured output directory.

2.3 Supported Package Managers

Out of the box, platform migrator supports conda as a package manager on any OS. If you wish to use platform migrator only with conda, you can skip this section. For Linux distros, pip, pacman, apt-get and aptitude are also supported out of the box. For Mac OS, pip is supported out of the box. For Windows, pip is supported in a POSIX shell environment like Cygwin.

In case of pip, pip2 and pip3 options are provided as well to explicitly use pip for a specific Python version. Using pip as the package manager will default to whichever version of pip is installed as the default. The other version of pip must be installed before it can be used.

However, platform migrator allows you to configure your own package manager if you do not wish to use any of the package managers listed above. See *Package manager config file* and *Configuring an alternate package manager* for complete instructions for that.

2.4 Git repo and other links

Git repository: https://gitlab.com/mmc691/platform-migrator
Online Documentation: https://platform-migrator.readthedocs.io

In case of bugs or questions, please create an issue through Gitlab.

CHAPTER

THREE

COMMAND LINE INTERFACE (CLI)

3.1 Command line arguments and options

On a target system, platform migrator executes in server mode or in migrate mode. In server mode, it accepts commands to control the HTTP server, interacts with the base system to receive the software and prepares it for migration. In migrate mode, it tries to migrate a software that has been prepared using either the server mode or from a zip file via the `pack` and `unpack` commands described below.

Additionally `pack` and `unpack` commands are available that can be used to pack and unpack a zip file for transfer over email or some means other than the platform migrator server. The `pack` command is run on the base system and it executes the client side of the server mode, while the `unpack` command is run on the target system to prepare the software for migration. These are helper commands which can be used as a replacement for the server mode, when it is not possible to connect the two systems over a network.

The usage for server mode is

```
platform-migrator server [-h] [--host HOST] [--port PORT] {start,stop,status}

commands:
  start                 Starts a simple http server on specified host and port
  stop                  Stops the currently running server
  status                Prints whether a server is currently running or not

optional arguments:
  -h, --help            show this help message and exit
  --host HOST           Hostname or IP address to listen on (Default: localhost)
  --port PORT           Port number to listen on (Default: 9000)
```

The usage for migrate mode is

```
platform-migrator migrate [-h] [--skip-tests] [--pck-mgr-configs PCK_MGR_CONFIGS]␣
↪name test_files [test_files ...]

positional arguments:
  name                   Name of package to migrate
  test_files             One or more test configuration files for migration

optional arguments:
  -h, --help             show this help message and exit
  --skip-tests           Skip tests but install dependencies and copy files
  --pck-mgr-configs PCK_MGR_CONFIGS
                         Config file for additional package managers
```

The usage for pack mode is

```
platform-migrator pack [-h] [-n NAME] [-d DIR]

Pack the software in a zip file for emailing

optional arguments:
  -h, --help            show this help message and exit
  -n NAME, --name NAME  Name of the software to migrate
  -d DIR, --dir DIR     Directory containing the software
```

The usage for unpack mode is

```
platform-migrator unpack [-h] zip_file

Unpack a zip file

positional arguments:
  zip_file    The zip file to unpack

optional arguments:
  -h, --help  show this help message and exit
```

Additionally, when receiving the script from the server to run, the script takes the same command line as `platform-migrator pack`. It can be executed as

```
python script.py [-h] [-n NAME] [-d DIR]

Transfer the software and additional data to the target system

optional arguments:
  -h, --help            show this help message and exit
  -n NAME, --name NAME  Name of the software to migrate
  -d DIR, --dir DIR     Directory containing the software
```

3.2 Environment variables

Platform migrator uses CONDA_HOME or CONDA_ROOT_DIR environment variables to determine the location of conda installation. These should point to the top level directory under which conda is installed. So, if conda is installed in the home directory of user bob, the value for either variable should be `/home/bob/anaconda`.

CHAPTER

FOUR

TUTORIALS

The following tutorials describe some use cases of platform migrator in moving software from one system to another.

4.1 Migrating from conda to conda

This is the simplest and most straightforward use case of platform migrator. The requirements for this use case are:

Software to be migrated

- The software must be in an executable format (either binary or source code)
- There should be a set of tests that can be run to determine whether the software works as intended or not.

Base System

- The base system must contain a dedicated conda environment for the software. This may be the conda root environment.
- Access to internet

Target System

- The target system must be a system where the user can install or has already installed conda. The tutorial will cover installation of conda.
- Access to internet

4.1.1 Preparing the target system

First, install platform migrator on the target system as per the instructions in *Installation*. There is no need for any configuration in this use case.

Next, go to Anaconda installation page and install the version of conda for your target system. You can skip this if you already have conda installed on the target system.

4.1.2 Transferring the software

There are 2 ways to do this. The end result of both the ways is the same and has no impact on the overall success or failure of the migration. Following any one of the two options below is sufficient to attempt an migration.

Option 1. Over a local network

On the target system, start the platform migrator server by executing

```
platform-migrator server --host 0.0.0.0 start
```

Here, platform migrator runs in server mode and starts a daemon process, which is a simple HTTP server which listens on all IP addresses and port 9001. If you wish to listen only on a particular IP address and port, use the `--host` and `--port` options to change them. Regardless, you should note the IP address of your target system. On Linux or Mac, this can be obtained by running `ifconfig` in a terminal and using the IP address on an interface other than the loopback interface, `lo`. On Windows, this can be obtained by running cmd and typing in `ipconfig`. Note the IP address for use on the base system.

Now, on the base system, download a python script from the target system by running

```
curl http://<IP-address-of-target-system>:9001/migrate > script.py
```

If curl is not available, activate the conda root environment on the base system and run the following code in a python interpreter

```
import requests
resp = requests.get('http://<IP-address-of-target-system>:9001/migrate')
with open('script.py', 'w') as fp:
    fp.write(resp.text)
exit()
```

The script is a Python script generated on the target system and it contains some procedures to probe the conda environment for the packages installed and to transfer the software and settings over to the target system.

Make sure that the conda executable is on the PATH of the shell. If it is not, run `export PATH=$PATH:/path/to/conda/bin` to add it to the PATH variable. Now, activate the conda environment that is used by the software to be migrated and run `python script.py` and follow the prompts on screen. The script will ask you to enter the name of the software and the directory under which the source code is saved, if everything goes well. When prompted regarding saving a zip file for emailing later, enter `n`. The prompts can be avoided by passing the name of the of the software and directory using command line options `-n` and `-d` respectively.

Option 2. With email, scp or other manual methods

This an alternate way to transfer software, in case network connection cannot be setup between the base and the target system. In this case, the server startup is not required on the target system and server mode is not involved.

To do this, first install platform migrator on the base system, by following the installation steps. After that, execute `platform-migrator pack`. This will run the same script as above, but when prompted to save a zip file say `y`. A zip file will be saved in the current directory. This can be copied to the target system using any method like email, scp, ftp or manual copy, and unpacked there using `platform-migrator unpack`. The prompts can be avoided by passing the name of the of the software and directory using command line options `-n` and `-d` respectively.

4.1.3 Installing and testing on the target system

Open up a text editor and create a test configuration for the software. A minimal test configuration would look like

```
[<software-name>]
package_managers = conda
output_dir = <directory-where-you-want-to-save-the-software>
tests = <test-name0>, <test-name1>
```

(continues on next page)

(continued from previous page)

```
[test_<test-name0>]
exec = <command-to-run-test0>

[test_<test-name1>]
exec = <command-to-run-test1>
```

The `<software-name>` should be replaced by the name used on the base system. The `output_dir` should be an absolute path to the directory where the software must be saved post migration. The `tests` option contains a comma separated list of tests that should be run on the software. For each test in this list, there must be a corresponding `test_<name>` section which contains a `exec` option containing the command to execute. This command must return 0 on success and any other value on failure. The command will be run from inside the directory in which software is installed and can use paths relative to that.

So, for example, a test called `foo` executes a script called `tests/foo.sh` inside the software, the test configuration will look like

```
[<software-name>]
package_managers = conda
output_dir = /tmp
tests = foo

[test_foo]
exec = bash tests/foo.sh
```

See *Writing test configuration files* for the complete documentation on this.

Once the tests have been configured, execute

```
platform-migrator migrate <software-name> <tests-config-file-name>
```

Now, platform migrator will run in `migrate` mode and try to install the software along with any required dependencies.

Platform migrator will create a new conda environment called <software-name> and run the tests for the software there. If the tests pass, the software will be copied into the output directory. Otherwise, the error message from the tests will be displayed on screen and the environment will be destroyed.

4.1.4 Installing without any tests

In case there are not tests available, you use the `--skip-tests` option and platform migrator will create a conda environment and copy the files to the output directory. A configuration file is still required but it need not contain any tests.

4.2 Configuring an alternate package manager

This tutorial walks you through another essential part of platform migrator, which is configuring it to use a package manage other than conda. These additional package managers can be added to the default file in `~/.platform-migrator/config/` on POSIX OSes. For Windows, replace ~ with the user's home directory.

Open up the `package-managers.ini` file in a text editor. The file contains the documentation of the file along with some sample package managers. These package managers are supported out of the box on Linux distros, but not on Windows and Mac. On Windows, you will probably need Cygwin or some other POSIX shell utility for these to be actually useful.

To add a new package manager to the file, create a new section. The section name will be used in the list of package managers in the test configuration file during the migration process, so keep the indicative of the package manager. Typically, this would be the name of the executable of the package manager.

In the section, add the following options and their values as described:

name The actual name. This is only for your benefit and not used by platform migrator.

exec The executable of the package manager used on the command line

install The command used to install a package where the package name will be at the end of the command.

install_version The command used to install a specific version of package.

search The command used to search the package manager for packages. This will typically require some further text manipulation since most package managers tend to provide human readable output rather than machine parseable output.

result_fmt A regular expression describing the results of the search.

Read the *Package manager config file* for details on how the values of each option should be written. An example of *Configuring apt-get* is also provided in the *Case Studies* section.

If you do not wish to edit the default file, you can save your configuration in a separate .ini file in the `~/.platform-migrator` directory and use it in addition to the default file by using the `--pck-mgr-configs ~/.platform_migrator/<filename>.ini` option during migration.

4.3 Migrating from conda to a different package manager(s)

This is a more complex migration which requires some input and knowledge on the user's end about the software being migrated. The requirements for this use case are:

Software to be migrated

- The software must be in an executable format (either binary or source code)
- There should be a set of tests that can be run to determine whether the software works as intended or not.

Base System

- The base system must contain a dedicated conda environment for the software. This may be the conda root environment.
- Access to internet

Target System

- The target system must contain one or more package managers and platform migrator must be configured to use them.
- Access to internet

4.3.1 Preparing the target system

First, install platform migrator on the target system as per the instructions in *Installation*. Follow the tutorial in *Configuring an alternate package manager* and configure a new package manager. Further instructions will assume that the package manager has been configured in a separate file called `~/.platform-migrator/site-pkg-mgrs.ini`

4.3.2 Transferring the software

This part is exactly the same as in the migration from conda to conda. Please refer the *Transferring the software* section for the documentation.

4.3.3 Installing and testing on the target system

Open up a text editor and create a test configuration for the software. A minimal test configuration would look like

```
[<software-name>]
package_managers = <site-package-manager0>, <site-package-manager1>
output_dir = <directory-where-you-want-to-save-the-software>
tests = <test-name0>, <test-name1>

[test_<test-name0>]
exec = <command-to-run-test0>

[test_<test-name1>]
exec = <command-to-run-test1>
```

The `package_managers` option is a comma-separated list of the package managers that will be used by platform-migrator for installing the required packages. The package managers are used in the order entered, so `<site-package-manager0>` is probed before probing `<site-package-manager1>`. The `<software-name>` should be replaced by the name used on the base system. The `output_dir` should be an absolute path to the directory where the software must be saved post migration. The `tests` option contains a comma separated list of tests that should be run on the software. For each test in this list, there must be a corresponding `test_<name>` section which contains a `exec` option containing the command to execute. This command must return 0 on success and any other value on failure. The command will be run from inside the directory in which software is installed and can use paths relative to that.

So, for example, a test called `foo` executes a script called `tests/foo.sh` inside the software and the site runs Archlinux with pip installed on it, the test configuration will look like

```
[<software-name>]
package_managers = pacman, pip
output_dir = /tmp
tests = foo

[test_foo]
exec = bash tests/foo.sh
```

Once the tests have been configured, execute

```
platform-migrator migrate --pck-mgr-configs site-pkg-mgrs.ini <software-name> <tests-
↪config-file-name>
```

Now, platform migrator will run in `migrate` mode and try to install the software along with any required dependencies.

Platform migrator will prompt the user for each package that needs to be installed based on the conda environment. For each package, there will be an option to skip the installation using that package manager. In this case, the next package manager will be probed, or the package will not be installed in case there are no further package managers to probe. If a package is not found in any of the package managers, the user will be asked whether the installation should be continued or not. After all packages are selected for installation, each package will be installed individually. Hence, a failure in the installation of one package will not prevent other packages from being installed. Once all the packages are installed, the tests will be executed and if successful, the software will be copied to the output directory.

4.3.4 Installing without any tests

In case there are not tests available, you use the `--skip-tests` option and platform migrator will follow the above process for installing the packages and copy the software over to the output directory, if the installations were completed successfully.

CHAPTER

FIVE

PACKAGE MANAGER CONFIG FILE

A package manager is configured in a .ini file. One file can contain multiple package managers.

Each package manager is a section and must contain the following keys:

name The name of the package manager. This can be any string.

exec The package manager executable. If empty, it defaults to the package manager name.

install An install command which takes the package name as the argument. Wildcard %p can be used to indicate the position of package name in the string. Otherwise, it is assumed that the package name is to be added to the end.

install_version An install command which takes package name and version as arguments. Wildcards %p and %v can be used in a format string to indicate the position of the package and version respectively.

search A search command which takes the search expression an input. Wildcard %s can be used to indicate position of the search expression. Otherwise, it is assumes that the expression should go at the end.

result_fmt A Python regexp which will match individual lines in the output returned by the search command. Wildcards %p and %v must be used to indicate the package name and version.

The commands can have the following substitution parameters:

%e The package manager executable defined in the same section. It may be used only in the `search`, `install` or `install_version` options. If this is not present in the option, the command will be executed as is.

%p The package name. It must be used in the `install_version` and in `result_fmt` otherwise the options are invalid. It may also be used in the `install` option, otherwise it will be appended to the the `install` option by default. It is not valid in other options.

%s A search string. It may be used only in the `search` option and will be appended by default if not provided.

%v The package version. It must be used in the `install_version` otherwise it is invalid. It may also be used in the `result_fmt` option. It is not valid in any other option.

CHAPTER

SIX

WRITING TEST CONFIGURATION FILES

This describes how to write a test configuration file for a package migration. The test configuration can contain tests and settings for multiple software or multiple test configuration files can be used for a particular software.

In other words, multiple test configurations files are read by platform migrator as if they were one file and only those sections which are applicable to a particular software are used.

There are two main types of sections, software sections and test sections. Test sections have names starting with `test_`. All other sections are treated as software sections. A special section `[DEFAULT]` may be used to create some default options for each software like package manager names and output directory location.

6.1 Software sections

A software section is named after the software being migrated and contains the following options:

```
[<software-name>]
package_managers = <pkg-mgr0>[, <pkg-mgr1>, ...]
output_dir = /an/absolute/path
tests = <test0>[, <test1>, ...]
```

The `package_managers` options is a comma separated list of package managers that will be used for migrating that particular software. These package manager names must correspond to section names in the package manager configuration files being used for the migration.

The `output_dir` is an absolute path to the location where the software should be saved.

`tests` contains a comma separated list of tests for the software. They must be test section names without the `test_` prefix and the test sections must be in one of the test configuration files being used for the migration.

6.2 Test sections

A test section begins with `test_` and contains only one option, `exec`:

```
[test_<test0>]
exec = python some_script.py

[test_<test1>]
exec = ~/test-script.sh
```

The option must be a command that can be executed by the default shell or an executable shell script. Any paths in the command must be relative to the directory which contains the software being migrated or an absolute path. So, `some_script.py` should be directly inside the software's top level directory while `test-script.sh` is in the home directory.

If the command returns 0 upon execution, the test is considered successful, otherwise it is considered a failure. The output from the tests is always printed out on the terminal.

CHAPTER

SEVEN

CASE STUDIES

All cases below have been run using conda version 4.3 or above. The cases where the target system is not conda are only recommended for expert users since the system-wide packages may need to be installed and user needs to be aware of what packages to install.

For all case studies, `<ip-address>` should be replaced with the IP address of the target system and `<conda-prefix>` should be replaced with the directory under which anaconda or miniconda is installed (eg. `/home/joe/miniconda3`).

7.1 Hello World over network

This is a basic use case which demonstrates how you can migrate simple script from one platform to another over a network. The script used for migration is a simple Python script that prints 'Hello world' and is located at `tests/basic-migration/simple-app/simple_app.py` in the git repository.

Two different scenarios, one where the target system is conda and one where the target system is Archlinux were run. The base system used was an Ubuntu 16.04 OS running inside VirtualBox, but it can be any system running conda and bash shell. Miniconda3 was installed on the base system by following the instructions at the Anaconda installation page

7.1.1 Target system with conda

First, a conda environment was created on the base system with

```
<conda-prefix>/bin/conda create -n simple-app python=3
. <conda-prefix>/bin/activate simple-app
```

The script was created in the home directory with

```
mkdir -pv simple-app
echo "print('Hello world')" > simple-app/simple_app.py
```

On the target system, the platform migrator was installed and the server was started with

```
pip3 install --user git+ssh://git@gitlab.com/mmc691/platform-migrator.git
platform-migrator server --host <IP-address> start
```

Now, back on the base system, the script was transferred with

```
curl http://<IP-address>:9001/migrate > script.py
python script.py
```

When prompted, for the application name, `simple-app` was entered, and for the directory, `simple-app` was entered. A zip file was not created by entering `n` on the prompt and the software was transferred over the network.

Note: The `python script.py` command can also be run as `python script.py -n simple-app -d simple-app/` to avoid the prompts.

After this, on the target system, a test configuration was created with the below contents and saved as `test-config.ini`

```
cat > test-config.ini << EOF
[simple-app]
package_managers = conda
output_dir = /tmp
tests = hello-world

[test_hello-world]
exec = python simple_app.py
EOF
```

Note: If you are trying to reproduce the case, you will have to manually create this file in a text editor. It is not generated by platform migrator.

Platform migrator was then executed with the following command

```
CONDA_HOME=<conda-prefix> platform-migrator migrate simple-app test-config.ini
```

It was verified that the package is available in `/tmp` by executing

```
ls -l /tmp
ls -l /tmp/simple-app
```

7.1.2 Target system with pacman

The initial steps of installing conda on the base system and platform migrator on the target system were completed as described above. The script was also transferred to the base system using curl command and was executed. However, on being prompted for the application name, `simple-app-pacman` was entered to avoid conflicts with the previous case. On the target system, the previously created `test-config.ini` was edited and the following content was added to it

```
[simple-app-pacman]
package_managers = pacman
output_dir = /tmp
tests = hello-world
```

Platform migrator was then executed with the following command

```
platform-migrator migrate simple-app-pacman test-config.ini
```

Since the script only requires Python, all packages other than `python` were skipped from installation. The tests were run and it was verified that the software was available in `/tmp` with

```
ls -l /tmp
ls -l /tmp/simple-app-pacman
```

This use case is available as a regression test in the git repository as `tests/basic-migration/test_simple_app.py` and the test config file is available as `tests/basic-migration/test-config.`

ini. The regression test uses two different conda environments on the local machine instead of a VM or a remote machine.

7.2 Hello World using zip file

This is a basic use case which demonstrates how you can migrate simple script from one platform to another using a zip file. The script used for migration is a simple Python script that prints 'Hello world' and is located at tests/basic-migration/simple-app/simple_app.py in the git repository.

Two different scenarios, one where the target system is conda and one where the target system is Archlinux were run. The base system used was an Ubuntu 16.04 OS running inside VirtualBox, but it can be any system running conda and bash shell. Miniconda3 was installed on the base system by following the instructions at the Anaconda installation page

7.2.1 Target system with conda

First, a conda environment was created on the base system with

```
<conda-prefix>/bin/conda create -n simple-app python=3
. <conda-prefix>/bin/activate simple-app
```

The script was created in the home directory with

```
mkdir -pv simple-app
echo "print('Hello world')" > simple-app/simple_app.py
```

Next, platform migrator was installed and the zip file for the software was created with

```
pip3 install --user git+ssh://git@gitlab.com/mmc691/platform-migrator.git
platform-migrator pack
```

When prompted, for the application name, simple-app was entered, and for the directory, simple-app was entered. A zip file was created by entering y on the prompt and it was copied manually to the target system.

Note: The platform-migrator pack command can also be run as platform-migrator pack -n simple-app -d simple-app/ to avoid the prompts.

Now, on the target system, platform migrator was installed and the software was setup for migration using

```
pip3 install --user git+ssh://git@gitlab.com/mmc691/platform-migrator.git
platform-migrator unpack simple-app.zip
```

After this, on the target system, a test configuration was created with the below contents and saved as test-config.ini

```
cat > test-config.ini << EOF
[simple-app]
package_managers = conda
output_dir = /tmp
tests = hello-world

[test_hello-world]
exec = python simple_app.py
EOF
```

Note: If you are trying to reproduce the case, you will have to manually create this file in a text editor. It is not generated by platform migrator.

Platform migrator was then executed with the following command

```
CONDA_HOME=<conda-prefix> platform-migrator migrate simple-app test-config.ini
```

It was verified that the package is available in `/tmp` by executing

```
ls -l /tmp
ls -l /tmp/simple-app
```

7.2.2 Target system with pacman

The initial steps of installing conda and platform migrator on the base system were completed as described above. A zip file was created once again by using the `platform-migrator pack` command and was copied to the target system. On being prompted for the application name, `simple-app-pacman` was entered to avoid conflicts with the previous case. Again as described above, platform migrator was installed on the target system and the software was unpacked using `platform-migrator unpack`.

On the target system, the previously created `test-config.ini` was edited and the following content was added to it

```
[simple-app-pacman]
package_managers = pacman
output_dir = /tmp
tests = hello-world
```

Platform migrator was then executed with the following command

```
platform-migrator migrate simple-app-pacman test-config.ini
```

Since the script only requires Python, all packages other than `python` were skipped from installation. The tests were run and it was verified that the software was available in `/tmp` with

```
ls -l /tmp
ls -l /tmp/simple-app-pacman
```

This use case is available as a regression test in the git repository as `tests/basic-migration/test_simple_app.py` and the test config file is available as `tests/basic-migration/test-config.ini`. The regression test uses two different conda environments on the local machine instead of a VM or a remote machine.

7.3 scikit-learn and scikit-image over network

Similar to the Hello World case, this case was run from an Ubuntu 16.04 VM system, once with conda and once with pacman and pip as the target package managers.

The script used for the software is available as `tests/basic-migration/scikit-app/scikit_app.py` in the git repo. It was saved on the base system as `~/scikit-app/scikit_app.py`.

7.3.1 Target system with conda

First, a conda environment was created on the base system with

```
<conda-prefix>/bin/conda create -n scikit-app python=2 scikit-learn scikit-image
. <conda-prefix>/bin/activate scikit-app
```

The script was created in the home directory with

```
git clone https://gitlab.com/mmc691/platform-migrator
cp -Rv platform-migrator/tests/basic-migration/scikit-app .
```

On the target system, platform migrator was installed and the server was started with

```
pip3 install --user git+ssh://git@gitlab.com/mmc691/platform-migrator.git
platform-migrator server --host <IP-address> start
```

Now, back on the base system, the script was transferred using

```
curl http://<IP-address>:9001/migrate > script.py
python script.py
```

When prompted, for the application name, `scikit-app` was entered, and for the directory, `scikit-app` was entered. A zip file was not created by entering `n` on the prompt and the software was transferred over the network.

Note: The `python script.py` command can also be run as `python script.py -n scikit-app -d scikit-app/` to avoid the prompts.

Since pip is listed before pacman, packages are first searched for in pip and only if the user decides not to install from pip, are they searched for in pacman. Also, pip2 was explicitly specified since it is known beforehand that the code works only for Python 2. This requires that pip2 is installed on the target system prior to using platform migrator.

Platform migrator was then executed with the following command

```
platform-migrator migrate scikit-app-pacman test-config.ini
```

If everything is successful, the application script will be installed under `/tmp`.

7.3.2 Target system with aptitude and pip

The initial steps of installing conda on the base system and platform migrator on the target system were completed as described above. The script was also transferred to the base system using curl command and was executed. However, on being prompted for the application name, `scikit-app-apt` was entered to avoid conflicts with the previous case. On the target system, the previously created `test-config.ini` was edited and the following content was added to it

```
[simple-app-apt]
package_managers = pip2, aptitude
output_dir = /tmp
tests = scikit-app
```

Since pip is listed before aptitude, packages are first searched for in pip and only if the user decides not to install from pip, are they searched for in aptitude.

Platform migrator was then executed with the following command

```
platform-migrator migrate scikit-app-apt test-config.ini
```

If everything is successful, the application script will be installed under `/tmp`.

The conda portion of this use case is available as a regression test in the git repository as `tests/basic-migration/test_scikit_app.py` and the test config file is available as `tests/basic-migration/test-config.ini`. The regression test uses two different conda environments on the local machine instead of a VM or a remote machine.

7.4 scikit-learn and scikit-image using zip file

This case study is exactly the same as the previous one, except the software was sent over to the target system as a zip file created using `platform-migrator pack`. This was also run from an Ubuntu 16.04 VM system, once with conda and once with pacman and pip as the target package managers.

The script used for the software is available as `tests/basic-migration/scikit-app/scikit_app.py` in the git repo. It was saved on the base system as `~/scikit-app/scikit_app.py`.

7.4.1 Target system with conda

First, platform migrator was installed on the base system with

```
pip3 install --user platform-migrator
```

Next, a conda environment was created on the base system (same as above) with

```
<conda-prefix>/bin/conda create -n scikit-app python=2 scikit-learn scikit-image
. <conda-prefix>/bin/activate scikit-app
```

The script was created in the home directory with

```
git clone https://gitlab.com/mmc691/platform-migrator
cp -Rv platform-migrator/tests/basic-migration/scikit-app .
```

Then, the a zip file was created using

```
platform-migrator pack
```

When prompted, for the application name, `scikit-app` was entered, and for the directory, `~/scikit-app` was entered. Here, a zip file was generated by entering `y` when prompted to create the file. The zip file was then emailed to the target system.

Note: The `platform-migrator pack` command can also be run as `platform-migrator pack -n scikit-app -d scikit-app/` to avoid the prompts.

After this, on the target system, a test configuration was created with the below contents and saved as `test-config.ini`

```
cat > test-config.ini << EOF
[scikit-app]
package_managers = conda
output_dir = /tmp
tests = scikit-app

[test_scikit-app]
exec = python2 scikit_app.py
EOF
```

Note: If you are trying to reproduce the case, you will have to manually create this file in a text editor. It is not generated by platform migrator.

Platform migrator was then executed with the following command

```
CONDA_HOME=<conda-prefix> platform-migrator migrate scikit-app test-config.ini
```

If everything is successful, the application script will be installed under `/tmp`.

7.4.2 Target system with pacman and pip

The initial steps of installing conda on the base system and platform migrator on the target system were completed as described above. The script was also transferred to the base system using curl command and was executed. However, on being prompted for the application name, `scikit-app-pacman` was entered to avoid conflicts with the previous case. On the target system, the previously created `test-config.ini` was edited and the following content was added to it

```
[simple-app-pacman]
package_managers = pip2, pacman
output_dir = /tmp
tests = scikit-app
```

On the target system, platform migrator was installed and the zip file was unpacked with

```
pip3 install --user git+ssh://git@gitlab.com/mmc691/platform-migrator.git
platform-migrator unpack <zip-file>
```

The rest of the steps are same as when transferring over a network, but are repeated here for the sake of completeness. After this, on the target system, a test configuration was created with the below contents and saved as `test-config.ini`

```
cat > test-config.ini << EOF
[scikit-app]
package_managers = conda
output_dir = /tmp
tests = scikit-app

[test_scikit-app]
exec = python2 scikit_app.py
EOF
```

Note: If you are trying to reproduce the case, you will have to manually create this file in a text editor. It is not generated by platform migrator.

Platform migrator was then executed with the following command

```
CONDA_HOME=<conda-prefix> platform-migrator migrate scikit-app test-config.ini
```

If everything is successful, the application script will be installed under `/tmp`.

7.4.3 Target system with pacman and pip

The initial steps of installing conda on the base system and platform migrator on the target system were completed as described above. The zip file was created and setup to the target system using pack and unpack commands. On being prompted for the application name, `scikit-app-pacman` was entered to avoid conflicts with the previous case.

On the target system, the previously created `test-config.ini` was edited and the following content was added to it

```
[simple-app-pacman]
package_managers = pip2, pacman
output_dir = /tmp
tests = scikit-app
```

Since pip is listed before pacman, packages are first searched for in pip and only if the user decides not to install from pip, are they searched for in pacman. Also, pip2 was explicitly specified since it is known beforehand that the code works only for Python 2. This requires that pip2 is installed on the target system prior to using platform migrator.

Platform migrator was then executed with the following command

```
platform-migrator migrate scikit-app-pacman test-config.ini
```

If everything is successful, the application script will be installed under `/tmp`.

7.4.4 Target system with aptitude and pip

The initial steps of installing conda on the base system and platform migrator on the target system were completed as described above. The zip file was created and setup to the target system using pack and unpack commands. On being prompted for the application name, `scikit-app-apt` was entered to avoid conflicts with the previous case. On the target system, the previously created `test-config.ini` was edited and the following content was added to it

```
[simple-app-apt]
package_managers = pip2, aptitude
output_dir = /tmp
tests = scikit-app
```

Since pip is listed before aptitude, packages are first searched for in pip and only if the user decides not to install from pip, are they searched for in aptitude.

Platform migrator was then executed with the following command

```
platform-migrator migrate scikit-app-apt test-config.ini
```

If everything is successful, the application script will be installed under `/tmp`.

The conda portion of this use case is available as a regression test in the git repository as `tests/basic-migration/test_scikit_app.py` and the test config file is available as `tests/basic-migration/test-config.ini`. The regression test uses two different conda environments on the local machine instead of a VM or a remote machine.

7.5 OpenAlea

This case was only done for conda to conda migration due to the complex dependencies required to install OpenAlea. The base system used was an Ubuntu 14.04 server with miniconda2 installed. The target system used was Archlinux with miniconda3 installed.

First, on the base system, a list of modules to install was obtained by searching the OpenAlea channel using

```
conda search -c OpenAlea --override-channels openalea.*
conda search -c OpenAlea --override-channels vplants.*
conda search -c OpenAlea --override-channels alinea.*
```

A conda environment was created on the base system with the above modules using the following commands

```
conda create -n openalea
. <conda-prefix>/bin/activate openalea
conda install -c OpenAlea <all-openalea-modules> <all-vplants-modules> <all-alinea-
↪modules>
```

A simple script was created to see if all packages can be imported. This script was used as the application to migrate.

```
cd ~
mkdir openalea
cat > openalea/test.py <<< EOF
try:
    import openalea, vplants, alinea
except ImportError:
    print('OpenAlea install failed')
    exit(-1)
else:
    print('OpenAlea installed succesfully')
    exit(0)
EOF
```

7.5.1 Transferring over network

When using the network to transfer the software, the server was started on the target system by executing `platform-migrator server --host <ip-address> start`

After that, the request was made to the platform-migrator server and the environment and the test script were sent over. The conda environment is still active on the base the system.

```
curl http://<ip-address>:9001/migrate > script.py
python script.py
```

The details were entered when prompted and a zip file was not created. After this, the steps from the *Installation and Testing* section below were executed.

Note: The `python script.py` command can also be run as `python script.py -n openalea -d openalea/` to avoid the prompts.

7.5.2 Transferring using zip file

When using a zip file to transfer the software, platform migrator was installed on the base system and the software was packed with `platform-migrator pack -n openalea -d openalea`.

The zip file was manually transferred to the target system using scp. Then, it was unpacked on the target system with `platform-migrator unpack openalea.zip`.

7.5.3 Installation and Testing

Now, on the target system, the `test-config.ini` file was created with

```
cat > test-config.ini << EOF
[openalea]
package_managers = conda
```

(continues on next page)

(continued from previous page)

```
output_dir = /tmp
tests = import_openalea

[test_import_openalea]
exec = python test.py
EOF
```

Note: If you are trying to reproduce the case, you will have to manually create this file in a text editor. It is not generated by platform migrator.

The migration was performed with

```
platform-migrator migrate openalea test-config.ini
```

If the conda environment was succesfully created, the test will pass otherwise it will fail.

7.6 Configuring apt-get

This case study describes how apt-get package manager was configured in the `config/package-managers.ini` file. This is the typical process that should be followed for configuring a new package manager.

First, a new section called `apt-get` was created in the config file. Then, the various options were added without any values.

```
[apt-get]
name =
exec =
install =
install_version =
search =
result_fmt =
```

The `name` and `exec` options were set to `apt-get` since the executable is called `apt-get`.

The installation of a new package is done using `sudo apt-get -y install <package-name>`. So, the `install` option was set to `sudo %e install`. The `%e` wildcard is replaced by the value in `exec` option and the package name is automatically append to the end of the command by platform migrator since the `%p` wildcard is not specified.

To install a specific version of the package in apt-get, `-` is used as a delimiter. So, the `install_version` option was set to `sudo %e apt-get %p-%v` and the `%p` and `%v` wildcards were used to specify the position of the package name and version.

Now, packages are searched using the `apt-cache` command. Since platform migrator searched using package name and version only, the `-n` flag is specified to `apt-cache search` so that package descriptions are not searched. However, the results from this still contain a small description for the package. So, the description needs to trimmed using other tools typically available on OSes which use `apt-get`. First, the results are piped into `awk` with the field delimiter set to `' - '`. The spaces around the hyphen make sure that the package name and description are split but the version and build string are part of the name. Only the name is printed out using `'{print $1}'` command. So, the search command now looks like `apt-cache search -n %s | awk -F ' - ' '{print $2}'`. Here, the `%s` wildcard replaced by the search expression.

Since the results are now in a format suitable to use for the install command, each result is treated as a separate package. So, the `result_fmt` option is set to `%p`. The final config section looks like

```
[apt-get]
name = apt-get
exec = apt-get
install = sudo %e -y install
install_version = sudo %e -y install %p-%v
search = apt-cache search -n %s | awk -F ' - ' '{print $2}'
result_fmt = %p
```

For other default package managers, a similar process was followed.

CHAPTER

EIGHT

PROCESS INTERNALS

8.1 Internal data and files

When platform migrator is installed, it creates an internal directory, `.platform_migrator` in the home directory of the user. This directory is stores all data related to the migrations attempted and also acts as the initial working directory for the process and the server. Platform migrator does not put any internal data outside this directory unless instructed to do so otherwise.

8.2 Starting the server

When the command `platform-migrator server start` is executed, the *`main`* module, switches the working directory to `.platform_migrator` and executes the script *`server`* as subprocess and exits.

The server script contains *`MigrateRequestHandler`* and *`PMServer`* classes, which are the request handler and the HTTP server respectively. The request handler implements a *`do_GET()`* method, which listens for get requests on the `/migrate` route and a *`do_POST()`*, which listens for any incoming data on `/yml`, `/min` and `/zip` routes. The `/yml` route is to receive the conda environment YAML file from the base system, `/min` for the minimal dependencies identified, if any, and `/zip` to receive a zip of the software itself.

All three POST routes take a JSON input with two attributes, `name` and `data`, where `name` contains the name of the software being migrated and `data` contains base64 encoded binary data corresponding to the what the route takes.

On startup, the server first creates a PIDFILE in the working directory, which is used to track which PID the server is running as, and also creates a copy of *`base_sys_script`* with the `HOST` and `PORT` variables updated to the value under which the server is running. This script is returned as the response when `/migrate` a request is received on `/migrate` route. The server now waits for requests.

The server will fail to start in case a PIDFILE already exists. So, only one instance of the server can be running at any point in time. When the server is stopped, it deletes the PIDFILE and the script before exiting.

8.3 Probing the base system

On the base system, platform migrator probes the conda environment using the functions in *`base_sys_script`*.

8.3.1 Receiving the probing script

This section is only run when transferring using the server. If transferring using a zip file, the script is executed using `platform-migrator pack` command.

The script gets downloaded to the base system when a request is made to the platform migrator server on the `/migrate` path. See the *Tutorials* for how to make the request using curl or Python. The script is compatible with both Python 2 and 3 specifically to allow it to run on older systems that may not be upgraded to latest version of Python.

When the script is executed, it first tries to identify which conda environment is active and uses that or prompts the user to enter the name of a conda environment and tries to source that before. In case the user input is required, it assumes that the `activate` shell script provided by conda is in the current working directory.

8.3.2 Handling transitive dependencies

Once the conda environment is identified, the script will run the `get_conda_min_deps()` function to identify a closure of conda packages that can be used to minimize the number of installs on the target system.

First, all the packages installed in the conda environment are obtained using `conda list` command. Then, for each dependency, a dry run of creating a new environment with that dependency as a target is done. This gives a list of packages that are required by the dependency. These packages are a second level dependency to the software being migrated and are removed from the list of direct dependencies of the software. Once all the dependencies have been processed, the remaining ones are the direct requirements for the software. This list is transferred over to the target system as well and only the packages in this list are installed on the target system. Their transitive dependencies are automatically installed by the package manager on the target system.

8.3.3 Collecting info

It then prompts the user to enter the name of the software and the directory where it is saved. The conda environment data, direct dependency list and the software are zipped up and sent back to the target system using the POST routes of the server. The script then exits.

8.4 Running a migration

Once the data from the base system has been transferred over, any number of migration attempts can be made on it using platform migrator. All data gets saved in `~/.platform_migrator/<software-name>/` directory on the target system. When `platform-migrator migrate <software-name> <test-config>` is executed, the `main` script parses the command line arguments and passes them over to the `migrate()` function, which is works as a wrapper to control the `Migrator` class.

The `Migrator` class parses the test config files and creates a new migration id for the job. This migration id is used to create a new directory for the migration and allows users to perform multiple attempts for the same software. In future, this may also store metadata about the attempted migration.

8.4.1 With conda as a package manager

Next, the conda environment YAML file is parsed and conda internal packages are removed. Now, if the test configuration lists conda as one of the package managers available, platform migrator will just use conda and ignore all other package managers. A new conda environment with the same name as the software is created and the YAML file is used to install the packages in it.

Now, the software is unzipped inside the migration directory and the tests configured for the software are run inside a sub-shell with the new conda environment activated. If multiple tests are configured, each test is run in a separate sub-shell. All tests are run even if one of the test fails. However, the migration is marked as an failure if any of the tests fail.

Once the migration is complete, the unzipped software is deleted. If the tests were successful, the software is unzipped again in the output directory from the test configuration.

8.4.2 With external package managers

If conda is not one of the package managers listed in the test configuration, `get_package_manager()` is used to parse the package manager configuration files. The function is a factory function for creating :py:class‘~platform_migrator.package_manager.PackageManager‘ objects. Once the package managers are obtained, each package from the direct dependency list of the software is searched for in the package managers and the user is prompted to confirm which of the search result should be used. If the user does not install any of the offered packages from any of the package managers, an option to abort the migration is offered.

The searches try to offer the user with as few options as possible by removing by using stricter search criteria first and only using the relaxed criteria if there are no results returned. As soon as a search returns results, they are presented to the user for selection.

Once all the packages have been selected by the user, they are installed one by one. This is intentionally done so that even if one package fails to install, other packages can still be installed and tests can be run.

The tests here are run similar to the conda case, except that there is no conda environment which needs to be activated. Other than that, the same process is used for running the tests.

CHAPTER NINE

FUTURE WORK

- Currently, platform migrator stores history of all migrations done, including failed attempts. However, the user cannot easily see this and the history is not used for anything. Work can be done around how this history can be used to determine systems and packages that are not compatible with each other. Also, features can be built around querying it.
- More package managers can be added and configured to be available by default. Also, there can be an option to list the usable package managers available on the system.
- Add integration for fetching software from Github, Gitlab and other git providers. Right now, the software needs to be fetched manually using `git clone` and then the conda environment needs to be setup using the `meta.yaml`. This can be automated in future iterations so that only the git repo URL is required.

CHAPTER

TEN

SOURCE CODE DOCUMENTATION

10.1 Main module

Main entry point for the package

This script parses the command line arguments and decides whether to run in server mode or migration mode.

`platform_migrator.main.`**`get_conda_prefix`**`()`

Adds the conda prefix to the environment as CONDA_HOME

`platform_migrator.main.`**`main`**`()`

Main function of the script

It sets the working directory, parses the command line arguments and calls the associated helper functions.

`platform_migrator.main.`**`parse_args`**`(`*cl_args*`)`

Parse the command line arguments

Args:

cl_args A list of arguments to parse

Return: An `argparser.Namespace` object containing the parsed arguments

`platform_migrator.main.`**`print_server_status`**`()`

Print whether server is running or not

`platform_migrator.main.`**`start_server`**`(`*host*, *port*`)`

Start the server as a subprocess

The function starts the server as a subprocess and then exits.

Args:

host The hostname or IP address to listen on

port The port number to listen on

`platform_migrator.main.`**`stop_server`**`()`

Stop the server

This is done by sending SIGTERM to the process running the server. The PID of the process is obtained from the PIDFILE

10.2 Server module

This module contains the HTTP server that communicates with the base system

The server is implemented in the `PMServer` class and runs as daemon, listening for requests on the `/migrate` path. The server can be controlled by running `platform-migrator server start` and `platform-migrator server stop`.

The module also contains the `MigrateRequestHandler` class which handles the requests. On recieving a request, the server returns a Python script that can be executed on the base system to start the migration process for a package. The Python script is in the `base_sys_script.py` file and the server's environment is updated whileresponding to a request.

The module is run as script when the application is started in server mode. It takes the following positional arguments:

host The hostname/IP address to listen on

port The post to listen on

working_directory The working directory of the server

class `platform_migrator.server.`**`MigrateRequestHandler`**(*request*, *client_address*, *server*)

Request handler for the PMServer class

The class processes each request recieved and sends the responses accordingly. It only supports GET requests on `/migrate` and POST requests on `/yml` and `/zip`.

`do_GET`()

Method to handle GET requests

The method responds with status 200 and the modified `base_sys_script.py` file as the response body on `/migrate` path. For all other paths, it responds with status code 400

`do_POST`()

Method to handle POST requests

The method responds with status 200 on `/yml`, `/min` and `/zip` paths if the request was processed without errors. Otherwise, it responds with status 400 in case of client error or status 500 in case of server error

class `platform_migrator.server.`**`PMServer`**(*host*, *port*)

A simple HTTP server that listens for requests for migration

It listens for new requests on `/migrate` path and the data for a request is received on `/yml` and `/zip` paths. The server is started using the `run()` method for the instance. The server creates a PIDFILE and starts a separate process to listen for requests.

Args:

host (str) The hostname or IP address to listen on

port (int) The port to listen on

`run`()

Start the server

`shutdown`(**args*)

Shutdown the server and exit

This method is a signal handler and is registered to SIGINT and SIGTERM during init.

10.3 Migrator module

This module contains the `Migrator` class which does a migration

The module also contains some helper functions to operate with the Migrator class, notably, the `migrate` function which migrates one or more packages

class `platform_migrator.migrator.`**`Migrator`**(*app_name*, *test_config_files*)

The Migrator class is basically a simple test suite runner

It reads the software config file provided, checks whether various packages are avialable or need to be installed and runs the test to verify whether the migration was succesful. It also stores any issues it encounters for easy look up later.

Args:

app_name The name of the software to migrate

test_config_files A list of filenames containing the test configuration for the package. This can be a single string for just one file or an iterable of strings for multiple files

`create_conda_env`()

Create a conda environment from a yml file

The environment is created using a subprocess and it is expected that the conda command is available on the PATH variable or at least one of CONDA_HOME and CONDA_ROOT_DIR environment variables is set to the prefix of the conda installation.

Return: True if the environment was created succesfully else False

`load_pck_mgrs`(*extra_pck_mgr_configs=None*)

Load the configuration options of various package managers

The method calls get_package_manager with the names of the package managers found in the package_managers option of the test config file.

Kwargs:

extra_pck_mgr_configs Any extra configuration files to read for obtaining the package manager's commands

`migrate_files`()

Copy application files to installation location

`run_migration`(*skip_tests=False*)

Run the migration process

The method setups the environment, runs the tests and if succesful, copies the files to the installation lcoation.

Kwargs

skip_test=False Skip running tests but perform the rest of the steps

`run_tests`()

Run the tests configured for the application

The tests are run in a shell which is started as a subprocess. All tests are run regardless of failures to provide as much information as possible in a single run.

Return: True if all the tests passed else False

`setup_env`()

Set up the environment for the application

This method parses the environment yml file and uses the package managers to install the required dependencies.

Return: A Boolean indicating whether all dependencies were succesfully installed

`platform_migrator.migrator.`**`get_target_from_search`**(*results*)

Prompt the user to select which package to install

This is a helper function for getting the user's preferred package to install from the packages that match the search criteria

Args:

results The results of the search. It is a list of dict-like objects where each item has the `'pck'` key and optionall a `'ver'` key for the package name and version respectively

Return: The user's selected package and version to install or None, if the user wants to skip the installation

`platform_migrator.migrator.`**`migrate`**(*name*, *test_files*, *pck_mgr_configs=None*, *skip_tests=False*)

Wrapper function to migrate one software

The function takes a list of test configuration files and attempts to migrate the package to the target system. All work is logged and saved and is used to determine any known incompatibilities.

Args:

name Name of the application to migrate

test_files An iterable of filenames which contain the configuration for testing a package

Kwargs:

pck_mgr_configs An iterable of filenames which contain additional package manage configurations

skip_test=False Skip running tests but perform the rest of the steps

10.4 Package manager module

This module contains the `PackageManager` class and related functions

It is used to search for and install packages on a given system. It also provides a helper function `get_package_manager` to instantiate a package manager from a .ini file.

The instances act as interfaces to the actual executables on the system and they parse and run the commands as configured. There is not real package manager implemented by platform migrator.

`class` `platform_migrator.package_manager.`**`PackageManager`**(*config*)

A package manager that can search and install packages

Each instance represents a package manager that is available on the system. It can search for packages and also install them, with user confirmation. It can also list any currently installed packages.

Args:

config A dict-like object that contains the commands to be executed for the various operations

Raises:

KeyError If any of the required keys is missing

ValueError If the inputs fail validation

`install`(*package*, *version=None*, *dry_run=False*)

Install a package on the system

The method will call the `install` or `install_version` command for the instance and try to install the package. In case version is not None, the specific version will be installed using the `install_version` command.

Args:

package (str) The name of the package to install

Kwargs:

version=None (str) A version string to install a specifc version of the package

dry_run=False (bool) If True, returns the command that will run without executing it.

Return: A Boolean indicating whether installation was succesful or not

Raises:

subprocess.CalledProcessError If the installation command fails

`parse_result`(*result*)

Parse the results of a search

The method is will take a string and try to parse it aa per the `result_fmt` format string defined for the instance.

Return: If succesful, the method returns a list of dictionaries which contain the parsed results

`search`(*expr*)

Search for packages using the `search` command

The method will search the package manager and return a list of the results found. The `exprs` argument are the search expressions that will be passed as an argument to the `search` command defined for the instance. The output must be in the format defined as per the `result_fmt` in the .

Args:

expr (str) An expression to search for

Return: A list of `Package` objects obtained from the search result

Raises:

ValueError If any of the results cannot be parsed

`platform_migrator.package_manager.`**`get_package_manager`**(*package_manager*, *extra_config_files=None*)

Create a new package manager from name

The function will read the default config file `conf/package-managers.ini` and any extra config files provided. If the package manager is defined as section in any of these files, a `PackageManager` object will be created and returned for that package manager.

Args:

package_manager The name of the package manager to use. It must be a section in one of the config files.

Kwargs:

extra_config_files=None A list of extra .ini files to read for the package manager configuration

Return: A new `PackageManager` object

Raises:

KeyError If the package manager was not defined in any of the .ini files parsed

`platform_migrator.package_manager.`**`parse_and_validate`**(*config*)

Parse and validate a package manager's configuration

The function checks that all the required keys are present and that the values for the options are valid.

Args:

config A dict-like objecct that contains the raw configuration options for a package manager

Return: A dict which contains the parsed and transformed commands for the package manager

Raises:

KeyError If any of the required keys is missing

ValueError If an input cannot be parsed or fails validation

10.5 Base system script

This module runs on the base system as a Python script

The module mainly discovers the existing conda environments and sends the source code over to the target system for further processing.

NOTE: The script should be compatible with both Python 2 and 3. It will be executed using whichever version of Python is available using `python` command.

`platform_migrator.base_sys_script.`**`create_zip_file`**(*zip_dir*, *env_yml=None*, *min_deps=None*, *quiet=False*)

Create a zip file of a directory

Args:

zip_dir The directory to zip

Kwargs:

env_yml=None (str) If provided, the string is copied as the `env.yml` file in the zip archive

min_deps=None (str) If provided, the string is copied as the `min-deps.json` file in the zip archive

quiet=False If True, no messages are printed

Return: A bytes buffer containing the zip file data

`platform_migrator.base_sys_script.`**`get_conda_env`**(*env=None*)

Return the yml file of current conda environment

The function runs a subprocess to get the yml file of the current conda environment. The results of the subprocess are returned as a tuple and if the yml file was obtained, it will be index 1 of the tuple.

Kwargs:

env=None Conda environment to source, if not the currently active environment.

Return: A tuple whose elements are a boolean indicating whether the process ran error free, the stdout of the process and the stderr of the process

`platform_migrator.base_sys_script.`**`get_conda_min_deps`**(*env*, *source_env=False*)

Try to create a list of minimal dependencies to install

The function queries the conda environment for all dependencies and then tries to determine the which dependencies are sub-dependencies of others. In this way, a minimal list of dependencies is created.

Args:

env The name of the conda environment to query

Kwargs:

source_env=False Boolean indicator whether to source conda env before getting data or not

Return: A list of conda package objects if succesful, or None if the conda envrionment could not be loaded

`platform_migrator.base_sys_script.`**`main`**`(`*cl_args=None*, *save_zip=None*`)`
Main function that is executed

Kwargs:

cl_args=None (argparse.Namespace) Any command line args that have been parsed.

save_zip=None (bool) If True, a zip file is created for saving without prompting the user. Otherwise, the user is prompted for the action.

`platform_migrator.base_sys_script.`**`parse_args`**`()`
Parse the command line arguments when run as a script

Return: An argparse.Namespace instance for the parsed arguments

`platform_migrator.base_sys_script.`**`send_data`**`(`*app_name*, *path*, *data*, *name*, *quiet=False*`)`
Send data back to the server

Args:

app_name (str) The name of the application

path (str) The path to send data to, without the base address

data Buffer containing the data

name (str) The name to use for the data in messages

Kwargs:

quiet=False (bool) If True, no messages are printed

`platform_migrator.base_sys_script.`**`send_env_yml`**`(`*app_name*, *env_yml*, *quiet=False*`)`
Send the environment yml data back to the server

Args:

app_name (str) The name of the application

env_yml Buffer containing the environment yml data

Kwargs:

quiet=False (bool) If True, no messages are printed

`platform_migrator.base_sys_script.`**`send_min_deps`**`(`*app_name*, *min_deps*, *quiet=False*`)`
Send a zip of the repository back to the server

Args:

app_name (str) The name of the application

min_deps (list) List containing the minimal required dependencies

Kwargs:

quiet=False (bool) If True, no messages are printed

`platform_migrator.base_sys_script.`**`send_repo_zip`**`(`*app_name*, *zip_file*, *quiet=False*`)`
Send a zip of the repository back to the server

Args:

app_name (str) The name of the application

zip_file Buffer containing the zip of the code repository

Kwargs:

quiet=False (bool) If True, no messages are printed

10.6 Unpack script

This module contains a helper function to unpack a zip file created by the `base_sys_script` into the right location and format

`platform_migrator.unpack.`**`unpack`**`(`*packed_zip*`)`

Unpack a zip file prepared by base_sys_script

This function unpacks the zip file, removes the env.yml and min-deps.json files from it and repacks the remaining files back as pkg.zip for use with the migrator

Args:

packed_zip (str) Absolute path to the zip file

INDEX