ReproZip Documentation

Release 1.1.0

Fernando Chirigati, Remi Rampin, Juliana Freire, and Dennis Sha

May 15, 2019

Contents

Welcome to ReproZip's documentation!

ReproZip is a tool aimed at simplifying the process of creating reproducible experiments from *command-line executions*. It tracks operating system calls and creates a package that contains all the binaries, files, and dependencies required to run a given command on the author's computational environment. A reviewer can then extract the experiment in his own environment to reproduce the results, even if the environment has a different operating system from the original one.

Currently, ReproZip can only pack experiments that originally run on Linux.

Concretely, ReproZip has two main steps:

- The *packing step* happens in the original environment, and generates a compendium of the experiment so as to make it reproducible. ReproZip tracks operating system calls while executing the experiment, and creates a `.rpz` file, which contains all the necessary information and components for the experiment.
- The *unpacking step* reproduces the experiment from the `.rpz` file. ReproZip offers different unpacking methods, from simply decompressing the files in a directory to starting a full virtual machine, and they can be used interchangeably from the same packed experiment. It is also possible to automatically replace input files and command-line arguments. Note that this step is also available on Windows and Mac OS X, since ReproZip can unpack the experiment in a virtual machine for further reproduction.

CHAPTER 1

Contents

1.1 Why ReproZip?

Reproducibility is a core component of the scientific process: it helps researchers all around the world to verify the results and to also build on them, allowing science to move forward. In natural science, long tradition requires experiments to be described in enough detail so that they can be reproduced by researchers around the world. The same standard, however, has not been widely applied to computational science, where researchers often have to rely on plots, tables, and figures included in papers, which loosely describe the obtained results.

The truth is computational reproducibility can be very painful to achieve for a number of reasons. Take the author-reviewer scenario of a scientific paper as an example. Authors must generate a compendium that encapsulates all the inputs needed to correctly reproduce their experiments: the data, a complete specification of the experiment and its steps, and information about the originating computational environment (OS, hardware architecture, and library dependencies). Keeping track of this information manually is rarely feasible: it is both time-consuming and error-prone. First, computational environments are complex, consisting of many layers of hardware and software, and the configuration of the OS is often hidden. Second, tracking library dependencies is challenging, especially for large experiments. If authors did not plan for reproducibility since the beginning of the project, reproducibility is drastically hampered.

For reviewers, even with a compendium in their hands, it may be hard to reproduce the results. There may be no instructions about how to execute the code and explore it further; the experiment may not run on his operating system; there may be missing libraries; library versions may be different; and several issues may arise while trying to install all the required dependencies, a problem colloquially known as dependency hell.

ReproZip helps alleviate these problems by allowing the user to easily capture all the necessary components in a single, distributable package. Also, the tool makes it easier to reproduce an experiment by providing different unpacking methods and interfaces that avoids the need to install all the required dependencies and that makes it possible to run the experiment under different inputs.

1.2 Installation

ReproZip is available as open source, released under the Revised BSD License. The tool is comprised of two components: **reprozip** (for the packing step) and **reprounzip** (for the unpack step). Additional components and plugins are also provided for *reprounzip*: **reprounzip-vagrant**, which unpacks the experiment in a Vagrant virtual machine; **reprounzip-docker**, which unpacks the experiment in a Docker container; and **reprounzip-vistrails**, which creates a VisTrails workflow to reproduce the experiment. More plugins may be developed in the future (and, of course, you are free to *roll your own*). In our website, you can find links to our PyPI packages and our GitHub repository.

In the following, you will find installation instructions for *Linux*, *Mac OS X*, and *Windows*. ReproZip is also available for the *Anaconda* Python distribution.

1.2.1 Linux

For Linux distributions, both *reprozip* and *reprounzip* components are available.

Required Software Packages

Python 2.7.3 or greater, or 3.3 or greater is required to run ReproZip[1]. If you don't have Python on your machine, you can get it from python.org. You will also need the pip installer.

Besides Python and pip, each component or plugin to be used may have additional dependencies that you need to install (if you do not have them already installed in your environment), as described below:

Component / Plugin	Required Software Packages
reprozip	SQLite, Python headers, a working C compiler
reprounzip	None
reprounzip-vagrant	Python headers, a working C compiler, SSL library[2], FFI library[2], Vagrant v1.1+, VirtualBox
reprounzip-docker	Docker
reprounzip-vistrails	None[3]

Debian and Ubuntu

You can get all the required dependencies using APT:

```
apt-get install python python-dev python-pip gcc libsqlite3-dev libssl-dev libffi-dev
```

Fedora & CentOS

You can get the dependencies using the Yum packaging manager:

```
yum install python python-devel gcc sqlite-devel openssl-devel libffi-devel
```

[1] `reprozip` and `reprounzip graph` will not work before 2.7.3 due to Python bug 13676 related to sqlite3. Python 2.6 is ancient and unsupported.

[2] Required to build PyCrypto.

[3] VisTrails v2.2.3+ is required to run the workflow generated by the plugin.

Installing *reprozip*

To install or update the *reprozip* component, simply run the following command:

```
$ pip install -U reprozip
```

Installing *reprounzip*

You can install or update *reprounzip* with all the available components and plugins using:

```
$ pip install -U reprounzip[all]
```

Or you can install *reprounzip* and choose components manually:

```
# Example, this installs all the components
$ pip install -U reprounzip reprounzip-docker reprounzip-vagrant reprounzip-vistrails
```

1.2.2 Mac OS X

For Mac OS X, only the *reprounzip* component is available.

Binaries

An installer containing Python 2.7, *reprounzip*, and all the plugins can be downloaded from GitHub.

Required Software Packages

Python 2.7.3 or greater, or 3.3 or greater is required to run ReproZip[4]. If you don't have Python on your machine, you can get it from python.org; you should prefer a 2.x release to a 3.x one. You will also need the pip installer.

Besides Python and pip, each component or plugin to be used may have additional dependencies that you need to install (if you do not have them already installed in your environment), as described below:

Component / Plugin	Required Software Packages
reprounzip	None
reprounzip-vagrant	Python headers, a working C compiler[5], Vagrant v1.1+, VirtualBox
reprounzip-docker	Docker
reprounzip-vistrails	None[6]

You will need Xcode installed, which you can get from the Mac App Store, and the Command Line Developer Tools; instrucions on installing the latter may depend on your Mac OS X version (some information on StackOverflow here).

See also:

Why does reprounzip-vagrant installation fail with error "unknown argument: -mno-fused-madd" on Mac OS X?

[4] `reprozip` and `reprounzip graph` will not work before 2.7.3 due to Python bug 13676 related to sqlite3. Python 2.6 is ancient and unsupported.

[5] Required to build PyCrypto.

[6] VisTrails v2.2.3+ is required to run the workflow generated by the plugin.

Installing *reprounzip*

First, be sure to upgrade *setuptools*:

```
$ pip install -U setuptools
```

You can install or update *reprounzip* with all the available components and plugins using:

```
$ pip install -U reprounzip[all]
```

Or you can install *reprounzip* and choose components manually:

```
# Example, this installs all the components
$ pip install -U reprounzip reprounzip-docker reprounzip-vagrant reprounzip-vistrails
```

1.2.3 Windows

For Windows, only the *reprounzip* component is available.

Binaries

A 32-bit installer containing Python 2.7, *reprounzip*, and all the plugins can be downloaded from GitHub.

Required Software Packages

Python 2.7.3 or greater, or 3.3 or greater is required to run ReproZip[7]. If you don't have Python on your machine, you can get it from python.org; you should prefer a 2.x release to a 3.x one. You will also need the pip installer.

Besides Python and pip, each component or plugin to be used may have additional dependencies that you need to install (if you do not have them already installed in your environment), as described below:

Component / Plugin	Required Software Packages
reprounzip	None
reprounzip-vagrant	PyCrypto[8], Vagrant v1.1+, VirtualBox
reprounzip-docker	Docker
reprounzip-vistrails	None[9]

See also:

Why does reprounzip-vagrant installation fail with error "Unable to find vcvarsall.bat" on Windows?

Installing *reprounzip*

You can install or update *reprounzip* with all the available components and plugins using:

```
$ pip install -U reprounzip[all]
```

Or you can install *reprounzip* and choose components manually:

[7] `reprozip` and `reprounzip graph` will not work before 2.7.3 due to Python bug 13676 related to sqlite3. Python 2.6 is ancient and unsupported.

[8] A working C compiler is required to build PyCrypto. For installation without building from source, please see this page.

[9] VisTrails v2.2.3+ is required to run the workflow generated by the plugin.

```
# Example, this installs all the components
$ pip install -U reprounzip reprounzip-docker reprounzip-vagrant reprounzip-vistrails
```

1.2.4 Anaconda

reprozip and *reprounzip* can also be installed on the Anaconda Python distribution, from anaconda.org:

```
$ conda install --channel vida-nyu reprozip reprounzip reprounzip-docker reprounzip-
↪vagrant reprounzip-vistrails
```

Note, however, that *reprozip* is only available for Linux.

1.3 Using *reprozip*

The *reprozip* component is responsible for packing an experiment, which is done in three steps: *tracing the experiment*, *editing the configuration file* (if necessary), and *creating the reproducible package*. Each of these steps is explained in more details below. Please note that *reprozip* is only available for Linux distributions.

1.3.1 Tracing an Experiment

First, *reprozip* needs to trace the operating system calls used by the experiment, so as to identify all the necessary information for its future re-execution, such as binaries, files, library dependencies, and environment variables.

The following command is used to trace a command line, or a *run*, used by the experiment:

```
$ reprozip trace <command-line>
```

where <*command-line*> is the command line. By running this command, *reprozip* executes <*command-line*> and uses `ptrace` to trace all the system calls issued, storing them in an SQLite database.

If you run the command multiple times, *reprozip* might ask you if you want to continue with your current trace (append the new command-line to it) or replace it (throw away the previous command-line you traced). You can skip this prompt by using either the `--continue` or `--overwrite` flag, like this:

```
$ reprozip trace --continue <command-line>
```

Note that the final package will be able to reproduce any of the runs, and files shared by multiple runs are only stored once.

By default, if the operating system is based on Debian or RPM packages (e.g.: Ubuntu, CentOS, Fedora, ...), *reprozip* will also try to automatically identify the distribution packages from which the files come, using the available package manager of the system. This is useful to provide more detailed information about the dependencies, as well as to further help when reproducing the experiment. However, note that the `trace` command can take some time doing that after the experiment finishes, depending on the number of file dependencies that the experiment has. To disable this feature, users may use the flag `--dont-identify-packages`:

```
$ reprozip trace --dont-identify-packages <command-line>
```

The database, together with a *configuration file* (see below), are placed in a directory named `.reprozip-trace`, created under the path where the `reprozip trace` command was issued.

1.3.2 Editing the Configuration File

The configuration file, which can be found in `.reprozip-trace/config.yml`, contains all the information necessary for creating the experiment package. This file is generated by the tracer and drives the packing step.

It is very likely that you won't need to modify this file, as the automatically-generated one should be sufficient to create a working package. However, in some cases, you may want to edit it prior to the creation of the package to add or remove files used by your experiment. This can be particularly useful, for instance, to remove big files that can be obtained elsewhere when reproducing the experiment, to keep the size of package small, and also to remove sensitive information that the experiment may use. The configuration file can also be used to edit the main command line, to add or remove environment variables, and to edit information regarding input/output files.

The first part of the configuration file gives general information with respect to the experiment and its runs, including command lines, environment variables, working directory, and machine information. Also, each run has a unique identifier (given by `id`) that is consistently used for packing and unpacking purposes:

```
# Run info
version: <reprozip-version>
runs:
# Run 0
- id: <run-id>
  architecture: <machine-architecture>
  argv: <command-line-arguments>
  binary: <command-line-binary>
  distribution: <linux-distribution>
  environ: <environment-variables>
  exitcode: <exit-code>
  gid: <group-id>
  hostname: <machine-hostname>
  system: <system-kernel>
  uid: <user-id>
  workingdir: <working-directory>

# Run 1
- id: ...
...
```

If necessary, users may change command line parameters by editing `argv`, and add or remove environment variables by editing `environ`. Users may also give a more meaningful and user-friendly identifier for a run by changing `id`. Other attributes should not be changed in general.

The next section brings information about input and output files, including their original paths and which runs read and/or wrote them. These are the files that *reprozip* identified as the main input or result of the experiment, which *reprounzip* will later be able to replace and extract from the experiment when reproducing it. You may add, remove, or edit these files in case *reprozip* fails in recognizing any important information, as well as give meaningful names to them by editing `name`:

```
# Inputs are files that are only read by a run; reprounzip can replace these
# files on demand to run the experiment with custom data.
# Outputs are files that are generated by a run; reprounzip can extract these
# files from the experiment on demand, for the user to examine.
# The name field is the identifier the user will use to access these files.
inputs_outputs:
  - name: <file-identifier>
    path: <path-to-file>
    read_by_runs: <run-ids>
    written_by_runs: <run-ids>
```

(continues on next page)

(continued from previous page)

```
  - name: ...
  ...
```

Note that you can prevent *reprozip* from identifying which files are input or output by using the `--dont-find-inputs-outputs` flag in the `reprozip trace` command.

Note: To visualize the dataflow of the experiment, pleaser refer to *Visualizing the Provenance Graph*.

See also:

Why doesn't 'reprozip' identify my input/output file?

The next section in the configuration file lists all the files to be packed. If the software dependencies were identified by the package manager of the system during the `reprozip trace` command, they will be organized in software packages and listed under `packages`; otherwise, file dependencies will be listed under `other_files`:

```
packages:
  - name: <package-name>
    version: <package-version>
    size: <package-size>
    packfiles: <include-package>
    files:
      # Total files used: <used-files-size>
      # Installed package size: <package-size>
      <files-list>
  - name: ...
  ...

other_files:
  <files-list>
```

The attribute `packfiles` can be used to control whether a software package will be packed: its default value is *true*, but users may change it to *false* to inform *reprozip* that the corresponding software package should not be included. To remove a file that was not identified as part of a package, users can simply remove it from the list under `other_files`.

Warning: Note that if a software package is requested not to be included, the *reprounzip* component will try to install it from a package manager when unpacking the experiment. If the software version from the package manager is different from (and incompatible with) the one used by the experiment, the experiment may not be reproduced correctly.

See also:

Why does 'reprounzip run' fail with "no such file or directory" or similar?

Last, users may add file patterns under `additional_patterns` to include other files that they think it will be useful for a future reproduction. As an example, the following would add everything under `/etc/apache2/` and all the Python files of all users from LXC containers (contrived example):

```
additional_patterns:
  - /etc/apache2/**
  - /var/lib/lxc/*/rootfs/home/**/*.py
```

Note that users can always reset the configuration file to its initial state by running the following command:

```
$ reprozip reset
```

Warning: When editing a configuration file, make sure your changes are as restrictive as possible, modifying only the necessary information. Removing important information and changing the structure of the file may cause issues while creating the package or unpacking the experiment.

1.3.3 Creating a Package

After tracing all the runs from the experiment and optionally editing the configuration file, the experiment package can be created by using the following command:

```
$ reprozip pack <package-name>
```

where <*package-name*> is the name given to the package. This command generates a `.rpz` file in the current directory, which can then be sent to others so that the experiment can be reproduced. For more information regarding the unpacking step, please see *Using reprounzip*.

Note that, by using `reprozip pack`, files will be copied from your environment to the package; as such, you should not change any file that the experiment used before packing it, otherwise the package will contain different files from the ones the experiment used when it was originally traced.

Warning: Before sending your package to others, it is advisable to test it and ensure that the reproduction of the experiment works.

1.3.4 Further Considerations

Packing Multiple Command Lines

As mentioned before, ReproZip allows multiple runs (i.e., command lines) to be traced and included in the same package. Alternatively, users can create a simple **script** that runs all the command lines, and pass *that* to `reprozip trace`. However, in this case, there will be no flexibility in choosing a single run to be reproduced, since the entire script will be re-executed.

Note that this flexibility has the caveat that users may reproduce the runs in a different order than the one originally used while tracing. If the order is important for the reproduction (e.g.: each run represents a step in a dataflow), please make sure to inform the correct reproduction order to whoever wants to replicate the experiment. This can also be obtained by running `reprounzip graph`; please refer to *Creating a Provenance Graph* for more information.

ReproZip can also combine multiple traces into a single one, in order to create a single package, using the `reprozip combine` command. The runs of each subsequent trace are simply appended in order.

Packing GUI and Interactive Tools

ReproZip is able to pack GUI tools. Additionally, there is no restriction in packing interactive experiments (i.e., experiments that require input from users). Note, however, that if entering something different can make the experiment load additional dependencies, the experiment will probably fail when reproduced on a different machine.

Capturing Connections to Servers

When reproducing an experiment that communicates with a server, the experiment will try to connect to the same server, which may or may not fail depending on the status of the server at the moment of the reproduction. However, if the experiment uses a local server (e.g.: database) that the user has control over, this server can also be captured, together with the experiment, to ensure that the connection will succeed. Users should create a script to:

- start the server,
- execute the experiment, and
- stop the server,

and use *reprozip* to trace the script execution, rather than the experiment itself. In this way, ReproZip is able to capture the local server as well, which ensures that the server will be alive at the time of the reproduction.

For example, if you have an web app that uses PostgreSQL and that runs until `Ctrl+C` is received, you can use the following script:

```
#!/bin/sh

/etc/init.d/postgresql start        # Start PostgreSQL

trap ' ' INT                        # Don't exit the whole script on Ctrl+C
./manage.py runserver 0.0.0.0:8000
trap - INT

/etc/init.d/postgresql stop         # Stop PostgreSQL
```

Note the use of `trap` to avoid exiting the entire script when pressing `Ctrl+C`, to make sure that the database gets shutdown via the next command.

Excluding Sensitive and Third-Party Information

ReproZip automatically tries to identify log and temporary files, removing them from the package, but the configuration file should be edited to remove any sensitive information that the experiment uses, or any third-party file/software that should not be distributed. Note that the ReproZip team is **not responsible** for personal and non-authorized files that may get distributed in a package; users should double-check the configuration file and their package before sending it to others.

Identifying Output Files

The *reprozip* component tries to automatically identify the main output files generated by the experiment during the `trace` command to provide useful interfaces for users during the unpacking step. However, if the experiment creates unique names for its outputs every time it is executed (e.g.: names with current date and time), the *reprounzip* component will not be able to correctly detect these; it assumes that input and output files do not have their path names changed between different executions. In this case, handling output files will fail. It is recommended that users modify their experiment (or use a wrapper script) to generate a symbolic link (with a fixed name) that always points to the latest result, and use that as the output file's path in the configuration file (under the `inputs_outputs` section).

1.4 Using *reprounzip*

While *reprozip* is responsible for tracing and packing an experiment, *reprounzip* is the component used for the unpacking step. *reprounzip* is distributed with three **unpackers** for Linux (*reprounzip directory*, *reprounzip chroot*, and

reprounzip installpkgs), but more unpackers are supported by installing additional plugins; some of these plugins are compatible with different environments as well (e.g.: *reprounzip-vagrant* and *reprounzip-docker*).

1.4.1 Inspecting a Package

Showing Package Information

Before unpacking an experiment, it is often useful to have further information with respect to its package. The `reprounzip info` command allows users to do so:

```
$ reprounzip info <package>
```

where **<***package***>** corresponds to the experiment package (i.e., the `.rpz` file).

The output of this command has three sections. The first section, *Pack information*, contains general information about the experiment package, including size and total number of files:

```
----- Pack information -----
Compressed size: <compressed-size>
Unpacked size: <unpacked-size>
Total packed paths: <number>
```

The next section, *Metadata*, contains information about dependencies (i.e., software packages), machine architecture from the packing environment, and experiment runs:

```
----- Metadata -----
Total software packages: <total-number-software-packages>
Packed software packages: <number-packed-software-packages>
Architecture: <original-architecture> (current: <current-architecture>)
Distribution: <original-operating-system> (current: <current-operating-system>)
Runs:
    <run-id>: <command-line>
    <run-id>: <command-line>
    ...
```

Note that, for *Architecture* and *Distribution*, the command shows information with respect to both the original environment (i.e., the environment where the experiment was packed) and the current one (i.e., the environment where the experiment is to be unpacked). This helps users understand the differences between the environments in order to provide a better guidance in choosing the most appropriate unpacker.

If the verbose mode is used, more detailed information on the runs is provided:

```
$ reprounzip -v info <package>
...
----- Metadata -----
...
Runs:
    <run-id>: <command-line>
        wd: <working-directory>
        exitcode: <exit-code>
    <run-id>: <command-line>
        wd: <working-directory>
        exitcode: <exit-code>
    ...
```

Last, the section *Unpackers* shows which of the installed *reprounzip* unpackers can be successfully used in the current environment:

```
----- Unpackers -----
Compatible:
    ...
Incompatible:
    ...
Unknown:
    ...
```

Compatible lists the unpackers that can be used in the current environment, while *Incompatible* lists the unpackers that are not supported in the current environment. When using the verbose mode, an additional *Unknown* list shows the installed unpackers that may not work. As an example, for an experiment originally packed on Ubuntu and a user reproducing it on Windows, the *vagrant* unpacker (available through the *reprounzip-vagrant* plugin) is compatible, but *installpkgs* is not; *vagrant* may also be listed under *Unknown* if `vagrant` is not in PATH (e.g.: if Vagrant is not installed).

Showing Input and Output Files

The `reprounzip showfiles` command can be used to list the input and output files defined for the experiment. These files are identified by an id, which is either chosen by ReproZip or set in the configuration file before creating the `.rpz` file:

```
$ reprounzip showfiles package.rpz
Input files:
    program_config
    ipython_config
    input_data
Output files:
    rendered_image
    logfile
```

Using the flag `-v` shows the complete path of each of these files in the experiment environment:

```
$ reprounzip -v showfiles package.rpz
Input files:
    program_config (/home/user/.progrc)
    ipython_config (/home/user/.ipython/profile_default/ipython_config.py)
    input_data (/home/user/experiment/input.bin)
Output files:
    rendered_image (/home/user/experiment/output.png)
    logfile (/home/user/experiment/log.txt)
```

You can use the `--input` or `--output` flags to show only files that are inputs or outputs. If the package contains multiple runs, you can also filter files for a specific run:

```
$ reprounzip -v showfiles package.rpz preprocessing-step
Input files:
    input_data (/home/user/experiment/input.bin)
Output files:
    logfile (/home/user/experiment/log.txt)
```

where *preprocessing-step* is the run id. To see the dataflow of the experiment, please refer to *Visualizing the Provenance Graph*.

The `reprounzip showfiles` command is particularly useful if you want to replace an input file with your own, or to get and save an output file for further examination. Please refer to *Managing Input and Output Files* for more information.

New in version 1.0.4: The `--input` and `--output` flags.

Creating a Provenance Graph

ReproZip also allows users to generate a *provenance graph* related to the experiment execution by reading the metadata available in the `.rpz` package. This graph shows the experiment runs as well as the files and other dependencies they access during execution; this is particularly useful to visualize and understand the dataflow of the experiment.

See *Visualizing the Provenance Graph* for details.

1.4.2 Unpackers

From the same `.rpz` package, *reprounzip* allows users to set up the experiment for reproduction in several ways by the use of different *unpackers*. Unpackers are plugins that have general interface and commands, but can also provide their own command-line syntax and options. Thanks to the decoupling between packing and unpacking steps, `.rpz` files from older versions of ReproZip can be used with new unpackers.

The *reprounzip* tool comes with three unpackers that are only compatible with Linux (`reprounzip directory`, `reprounzip chroot`, and `reprounzip installpkgs`). Additional unpackers, such as `reprounzip vagrant` and `reprounzip docker`, can be installed separately. Next, each unpacker is described in more details; for more information on how to use an unpacker, please refer to *Using an Unpacker*.

The *directory* Unpacker: Unpacking as a Plain Directory

The *directory* unpacker (`reprounzip directory`) allows users to unpack the entire experiment (including library dependencies) in a single directory, and to reproduce the experiment directly from that directory. It does so by automatically setting up environment variables (e.g.: PATH, HOME, and LD_LIBRARY_PATH) that point the experiment execution to the created directory, which has the same structure as in the packing environment.

Please note that, although this unpacker is easy to use and does not require any privilege on the reproducing machine, it is **unreliable** since the directory is not isolated in any way from the remainder of the system. In particular, should the experiment use absolute paths, they will hit the host system instead. However, if the system has all the required packages (see *The installpkgs Unpacker: Installing Software Packages*), and the experiment's files are addressed with relative paths, the use of this unpacker should not cause any problems.

> **Warning:** `reprounzip directory` provides no isolation of the filesystem, as mentioned before. If the experiment uses absolute paths, either provided by you or hardcoded in the experiment, **they will point outside the unpacked directory**. Please be careful to use relative paths in the configuration and command line if you want this unpacker to work with your experiment. Other unpackers are more reliable in this regard.

Note: `reprounzip directory` is automatically distributed with *reprounzip*.

See also:

Why does 'reprounzip directory' fail with "IOError"?

The *chroot* Unpacker: Providing Isolation with the *chroot* Mechanism

In the *chroot* unpacker (`reprounzip chroot`), similar to `reprounzip directory`, a directory is created from the experiment package; however, a full system environment is also built, which can then be run with

`chroot(2)`, a Linux mechanism that changes the root directory `/` for the experiment to the experiment directory. Therefore, this unpacker addresses the limitation of the *directory* unpacker and does not fail in the presence of hard-coded absolute paths. Note as well that it **does not interfere with the current environment** since the experiment is isolated in that single directory.

Warning: Do **not** try to delete the experiment directory manually; **always** use `reprounzip chroot destroy`. If `/dev` is mounted inside, you will also delete your system's device pseudo-files (these can be restored by rebooting or running the `MAKEDEV` script).

Note: Although *chroot* offers pretty good isolation, it is not considered completely safe: it is possible for processes owned by root to "escape" to the outer system. We recommend not running untrusted programs with this plugin.

Note: `reprounzip chroot` is automatically distributed with *reprounzip*.

The *installpkgs* Unpacker: Installing Software Packages

By default, ReproZip identifies if the current environment already has the required software packages for the experiment, then using the installed ones for reproduction. For the non-installed software packages, it uses the dependencies packed in the original environment and extracted under the experiment directory.

Users may also let ReproZip try and install all the dependencies of the experiment on their machine by using the *installpkgs* unpacker (`reprounzip installpkgs`). This unpacker currently works for distribution based on Debian or RPM packages (e.g.: Ubuntu, CentOS, Fedora, ...), and uses the package manager to automatically install all the required software packages directly on the current machine, thus **interfering with your environment**.

To install the required dependencies, the following command should be used:

```
$ reprounzip installpkgs <package>
```

Users may use flag *y* or *assume-yes* to automatically confirm all the questions from the package manager; flag *missing* to install only the software packages that were not originally included in the experiment package (i.e.: software packages excluded in the configuration file); and flag *summary* to simply provide a summary of which software packages are installed or not in the current environment **without installing any dependency**.

Warning: Note that the package manager may not install the same software version as required for running the experiment, and if the versions are incompatible, the reproduction may fail.

Note: This unpacker is only used to install software packages. Users still need to use either `reprounzip directory` or `reprounzip chroot` to extract the experiment and execute it.

Note: `reprounzip installpkgs` is automatically distributed with *reprounzip*.

The *vagrant* Unpacker: Building a Virtual Machine

The *vagrant* unpacker (`reprounzip vagrant`) allows an experiment to be unpacked into a Virtual Machine and reproduced in that emulated environment, by automatically using Vagrant. Therefore, the experiment can be reproduced in any environment supported by this tool, i.e., Linux, Mac OS X, and Windows. Note that the plugin assumes that Vagrant and VirtualBox are installed on your machine.

In addition to the commands listed in *Using an Unpacker*, you can use `suspend` to save the virtual machine state to disk, and `setup/start` to restart a previously-created machine:

```
$ reprounzip vagrant suspend <path>
$ reprounzip vagrant setup/start <path>
```

The `setup` command also takes a `--memory` argument to explicitely select how many megabytes of RAM to allocate to the virtual machine.

Note: This unpacker is **not** distributed with *reprounzip*; it is a separate package that should be installed before use (see *Installation*).

New in version 1.0.1: The `--memory` option.

New in version 1.0.4: The `suspend` command.

The *docker* Unpacker: Building a Docker Container

ReproZip can also extract and reproduce experiments as Docker containers. The *docker* unpacker (`reprounzip docker`) is responsible for such integration and it assumes that Docker is already installed in the current environment.

You can pass arguments to the `docker(1)` program by using the `--docker-option` option to the `setup` or `run` commands.

Thanks to Docker's image layers feature, you can easily go back to the initial image after having run commands in the environment or replaced input files. To do that, use the `reset` command:

```
$ reprounzip docker reset <path>
```

Note: This unpacker is **not** distributed with *reprounzip*; it is a separate package that should be installed before use (see *Installation*).

1.4.3 Using an Unpacker

Once you have chosen (and installed) an unpacker for your machine, you can use it to setup and run a packaged experiment. An unpacker creates an **experiment directory** in which the working files are placed; these can be either the full filesystem (for *directory* or *chroot* unpackers) or other content (e.g.: a handle on a virtual machine for the *vagrant* unpacker); for the *chroot* unpacker, it might have mount points. To make sure that you free all resources and that you do not damage your environment, you should **always use the destroy command** to delete the experiment directory, not just merely delete it manually. See more information about this command below.

All the following commands need to state which unpacker is being used (i.e., `reprounzip directory` for the *directory* unpacker, `reprounzip chroot` for the *chroot* unpacker, `reprounzip vagrant` for the *vagrant* unpacker, and `reprounzip docker` for the *docker* unpacker). For the purpose of this documentation, we will use the *vagrant* unpacker; to use a different one, just replace `vagrant` in the following with the unpacker of your interest.

See also:

Structure of Unpacked Experiments provides further detailed information on unpackers.

Setting Up an Experiment Directory

Note: Some unpackers require an Internet connection during the `setup` command, to download some of the support software or the packages that were not packed. Make sure that you have an Internet connection, and that there is no firewall blocking the access.

To create the directory where the execution will take place, the `setup` command should be used:

```
$ reprounzip vagrant setup <package> <path>
```

where <*path*> is the directory where the experiment will be unpacked, i.e., the experiment directory.

Note that, once this is done, you should only remove <*path*> with the *destroy* command described below: deleting this directory manually might leave files behind, or even damage your system through bound filesystems.

The other unpacker commands take the <*path*> argument; they do not need the original package for the reproduction.

Reproducing the Experiment

After creating the directory, the experiment can be reproduced by issuing the `run` command:

```
$ reprounzip vagrant run <path>
```

which will execute the experiment inside the experiment directory. Users may also change the command line of the experiment by using `--cmdline`:

```
$ reprounzip vagrant run <path> --cmdline <new-command-line>
```

where <*new-command-line*> is the modified command line. This is particularly useful to reproduce and test the experiment under different input parameter values. Using `--cmdline` without an argument only prints the original command line.

If the package contains multiple *runs* (separate commands that were packed together), all the runs are reproduced. You can also provide the id of the run or runs to be used:

```
$ reprounzip vagrant run <path> <run-id>
$ reprounzip vagrant run <path> <run-id> --cmdline <new-command-line>
```

For example:

```
$ reprounzip vagrant run unpacked-experiment 0-1,3  # First, second, and fourth runs
$ reprounzip vagrant run unpacked-experiment 2-  # Third run and up
$ reprounzip vagrant run unpacked-experiment compile,test  # Runs named 'compile' and
↪'test', in this order
```

If the experiment involves running a GUI tool, the graphical interface can be enable by using `--enable-x11`:

```
$ reprounzip vagrant run <path> --enable-x11
```

which will forward the X connection from the experiment to the X server running on your machine. In this case, make sure you have a running X server.

Note that in some situations, you might want to pass specific environment variables to the experiment, for example to set execution limits or parameters (such as OpenMPI information). To that effect, you can use the `--pass-env VARNAME` option to pass variables from the current machine, overriding the value from the original packing machine (*VARNAME* can be a regex). You can also set a variable to any value using `--set-env VARNAME=value`. For example:

```
$ reprounzip vagrant run unpacked-experiment --pass-env 'OMPI_.*' --pass-env LANG --
↪set-env DATA_SERVER_ADDRESS=localhost
```

New in version 1.0.3: The `--pass-env` and `-set-env` options.

Removing the Experiment Directory

The `destroy` command will unmount mounted paths, destroy virtual machines, free container images, and delete the experiment directory:

```
$ reprounzip vagrant destroy <path>
```

Make sure you always use this command instead of simply deleting the directory manually.

Managing Input and Output Files

When tracing an experiment, ReproZip tries to identify which are the input and output files of the experiment. This can also be adjusted in the configuration file before packing. If the unpacked experiment has such files, ReproZip provides some commands to manipulate them.

First, you can list these files using the `showfiles` command:

```
$ reprounzip showfiles <path>
Input files:
    program_config
    ipython_config
    input_data
Output files:
    rendered_image
    logfile
```

To replace an input file with your own, *reprounzip*, you can use the `upload` command:

```
$ reprounzip vagrant upload <path> <input-path>:<input-id>
```

where <*input-path*> is the new file's path and <*input-id*> is the input file to be replaced (from `showfiles`). This command overwrites the original path in the environment with the file you provided from your system. To restore the original input file, the same command, but in the following format, should be used:

```
$ reprounzip vagrant upload <path> :<input-id>
```

Running the `showfiles` command shows what the input files are currently set to:

```
$ reprounzip showfiles <path> --input
Input files:
    program_config
```

(continues on next page)

(continued from previous page)

```
        (original)
    ipython_config
        C:\Users\Remi\Documents\ipython-config
```

In this example, the input *program_config* has not been changed (the one bundled in the `.rpz` file will be used), while the input *ipython_config* has been replaced.

After running the experiment, all the generated output files will be located under the experiment directory. To copy an output file from this directory to another desired location, use the `download` command:

```
$ reprounzip vagrant download <path> <output-id>:<output-path>
```

where <*output-id*> is the output file to be copied (from `showfiles`) and <*output-path*> is the desired destination of the file. If an empty destination is specified, the file will be printed to stdout:

```
$ reprounzip vagrant download <path> <output-id>:
```

You can also omit the colon `:` altogether to download the file to the current directory under its original name:

```
$ reprounzip vagrant download <path> <output-id>
```

or even use `--all` to download every output file to the current directory under their original names.

Note that the `upload` command takes the file id on the right side of the colon (meaning that the path is the origin, and the id is the destination), while the `download` command takes it on the left side (meaning that the id is the origin, and the path is the destination). Both commands move data from left to right.

New in version 1.0.4: Allow `download <output-id>` (no explicit destination), and add `--all`.

New in version 1.1.0: Allow uploading and downloading any file via its full path, instead of a input/output file id.

See also:

Why can't 'reprounzip' get my output files after reproducing an experiment?

Running the Experiment in VisTrails

In addition to reproducing the experiment, you may want to edit its dataflow by inserting your own processes between and around the experiment steps, or even by connecting multiple ReproZip'd experiments. However, manually managing the experiment workflow (with the help of `reprounzip upload/download` commands) can quickly become painful.

To allow users to easily manage these workflows, *reprounzip* provides a plugin for the VisTrails scientific workflow management system, which has easy-to-use interfaces to run and modify a dataflow. See *VisTrails Plugin* for more information.

1.4.4 Further Considerations

Reproducing Multiple Execution Paths

The *reprozip* component can only guarantee that *reprounzip* will successfully reproduce the same execution path that the original experiment followed. There is no guarantee that the experiment won't need a different set of files if you use a different configuration; if some of these files were not packed into the `.rpz` package, the reproduction may fail.

1.5 Visualizing the Provenance Graph

Note: If you are using a Python version older than 2.7.3, this feature will not be available due to Python bug 13676 related to sqlite3.

To generate a *provenance graph* related to the experiment execution, the `reprounzip graph` command should be used:

```
$ reprounzip graph graphfile.dot mypackfile.rpz
```

where *graphfile.dot* corresponds to the graph, and *mypackfile.rpz* corresponds to the experiment package.

Alternatively, you can generate the graph after running `reprozip trace` without creating a `.rpz` package:

```
$ reprounzip graph [-d tracedirectory] graphfile.dot
```

The graph is outputted in the DOT language. You can use Graphviz to load and visualize the graph:

```
$ dot -Tpng graphfile.dot -o graph.png
```

It is also possible to output a JSON file with the flag `--json`.

1.5.1 Command-Line Options

Since an experiment may involve a significantly large number of file dependencies, `reprounzip graph` offers several command-line options to control what will be shown in the provenance graph, as described below. By default it includes all information available, which is often unreadable (see `fig-toobig`).

Filtering Out Files

Files can be filtered out using a regular expression[1] with the flag `--regex-filter`. For example:

- `--regex-filter /~[^/]*$`` will filter out files whose name begins with a tilde
- `--regex-filter ^/usr/share` will filter out `/usr/share` recursively
- `--regex-filter \.bin$` will filter out files with a `.bin` extension

These flags can be passed multiple times.

Replacing Filenames

Users can remap filenames using regular expressions[1] with the flag `--regex-replace`. This can be used to:

- simplify the graph by making filenames shorter,
- aggregate multiple files to a single node by mapping them to the same name, or
- fix programs that are using some type of cache or for which the wrong access was logged, such as Python's `.pyc` files.

Example:

[1] Anchoring regular expressions with `^` and `$` and escaping dots (`\.`) is recommended. For more information about regular expressions, please see here.

- `--regex-replace .pyc$ \.py` will replace accesses to bytecode cache files (.pyc) to the original source (.py)
- `--regex-replace ^/dev(/.*)?$ /dev` will aggregate all device files as a single path */dev*
- `--regex-replace ^/home/vagrant/experiment/data/(.*)\.bin data:\1` will simplify the paths to some data files

The flag `--aggregate` is a shortcut allowing users to aggregate all files beginning with a given prefix. For instance, `--aggregate /usr/somepath` will collapse all files under `/usr/somepath` (this is equivalent to `--regex-replace '^/usr/somepath' '/usr/somepath'`).

Both flags can be passed multiple times.

Controlling Levels of Detail

Users can control the levels of detail for each category of items in the provenance graph.

Software Packages

- `--packages file` will show all the files belonging to a package grouped under that package's name
- `--packages package` will show the package as a single item, not detailing the individual files that it contains
- `--packages drop` will entirely hide the packages, removing all their files from the graph
- `--packages ignore` will ignore the package identification, handling their files as if they had not been detected as being part of a package

Note that regex filters and replacements are applied beforehand, so files that are remapped to a package will be shown under that package name.

Processes

- `--processes thread` will show every process and thread
- `--processes process` will show every process and hide threads
- `--processes run` will show only one node for an experiment run, even if the run is composed by multiple processes and threads

Other Files

For files that are not part of a software package, or if `--packages ignore` is being used:

- `--otherfiles all` will show every file (unless filtered by `--regex-filter`)
- `--otherfiles io` will show only the input and output files, as identified in the configuration file
- `--otherfiles no` will ignore all the files

1.5.2 Common Recipes

- Full provenance graph (likely to be unreadable for most experiments, due to the large amount of information to be presented):

```
$ reprounzip graph graph.dot myexperiment.rpz
```

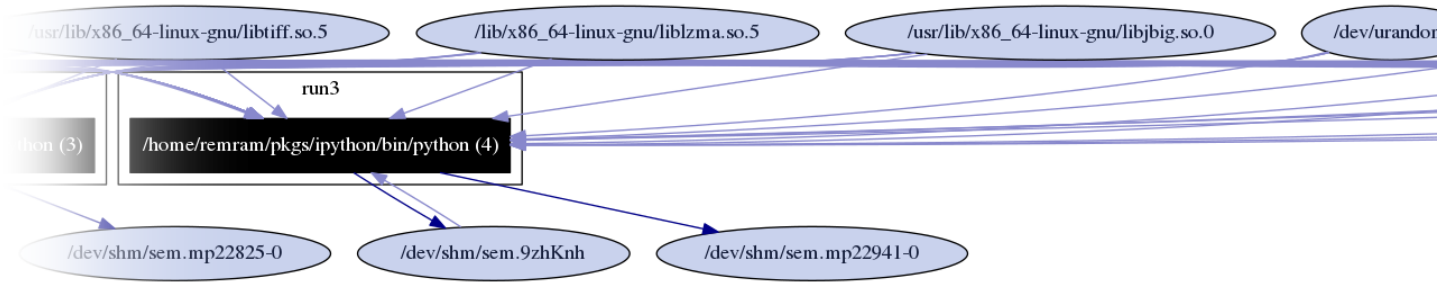


Fig. 1: Provenance graph showing all the information available (full graph). This represents the default configuration.

- Mapping Python bytecode cache files to their corresponding source file (this may help attribute file accesses to software packages):

```
$ reprounzip graph --regex-replace '\.pyc$' '\.py' graph.dot myexperiment.rpz
```

- Dataflow of the experiment, showing the runs and their corresponding input and output files:

```
$ reprounzip graph --packages drop --otherfiles io --processes run graph.dot␣
↪myexperiment.rpz
```

- Provenance graph showing only processes and threads (no file accesses):

```
$ reprounzip graph --packages drop --otherfiles drop --processes thread graph.dot␣
↪myexperiment.rpz
```

1.6 Making Jupyter Notebooks Reproducible with ReproZip

reprozip-jupyter is a plugin for Jupyter Notebooks, a popular open-source web application that allows you to create and share documents that contain live code, equations, visualizations and narrative text. These are valuable documents for data cleaning, analysis, writing executable papers/articles, and more. However, Jupyter Notebooks are subject to dependency hell like any other application – just the Notebook is not enough for full reproducibility. We have written a ReproZip plugin for Jupyter Notebooks to help users automatically capture dependencies (including data, environment variables, etc.) of Notebooks and also automatically set up those dependencies in another computing environment.

1.6.1 Installation

You can install *reprozip-jupyter* with pip:

```
$ pip install reprozip-jupyter
```

Or Anaconda:

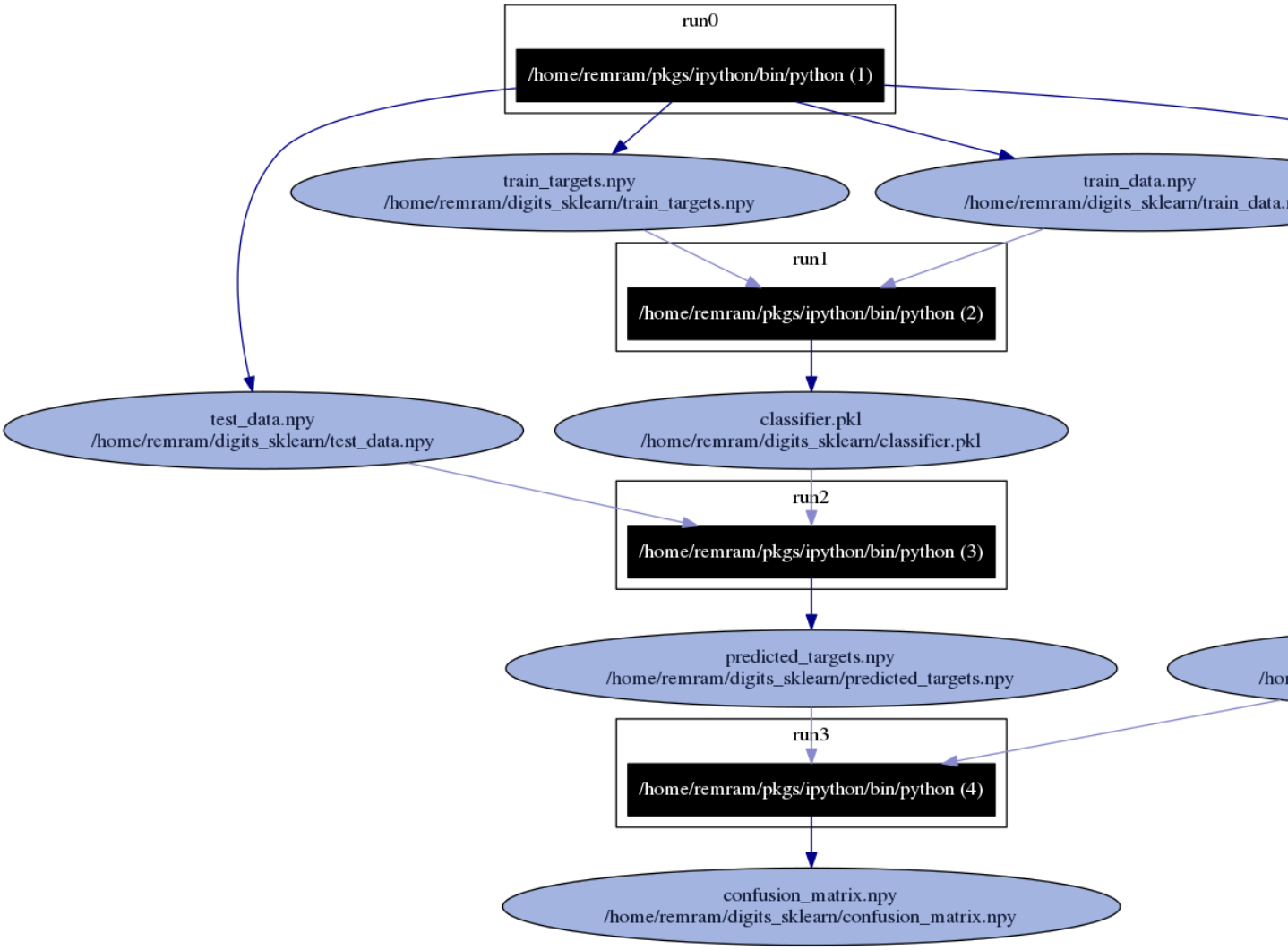


Fig. 2: Provenance graph showing input and output files for an experiment with 4 runs.

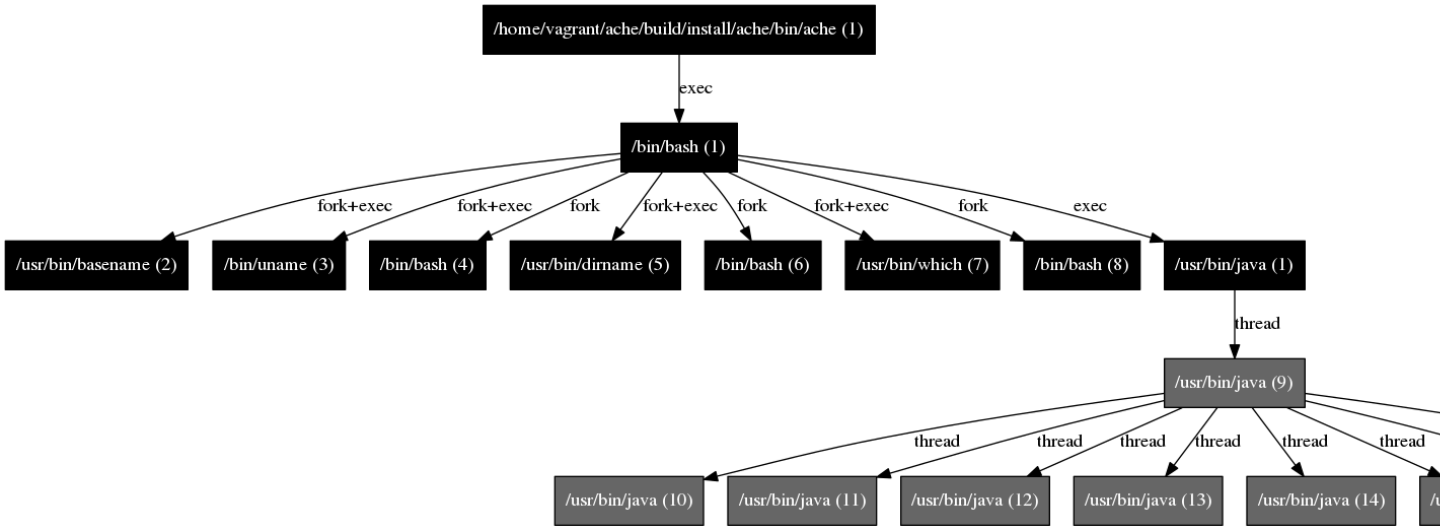


Fig. 3: Provenance graph showing only processes and threads.

```
$ conda install reprozip-jupyter
```

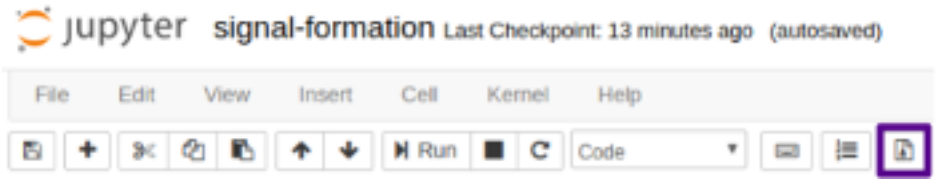
Once successfully installed, you should then enable the plugin for both the client and server side of Jupyter Notebooks:

```
$ jupyter nbextension install --py reprozip_jupyter --user
$ jupyter nbextension enable --py reprozip_jupyter --user
$ jupyter serverextension enable --py reprozip_jupyter --user
```

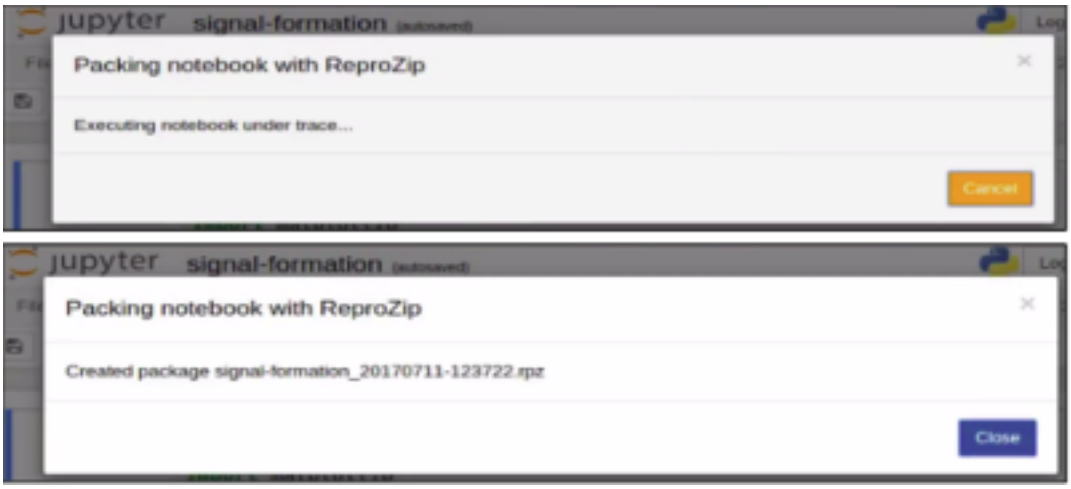
Once these steps are completed, when you start a Jupyter Notebook server, you should be able to see the ReproZip button in your notebook's toolbar.

1.6.2 Packing

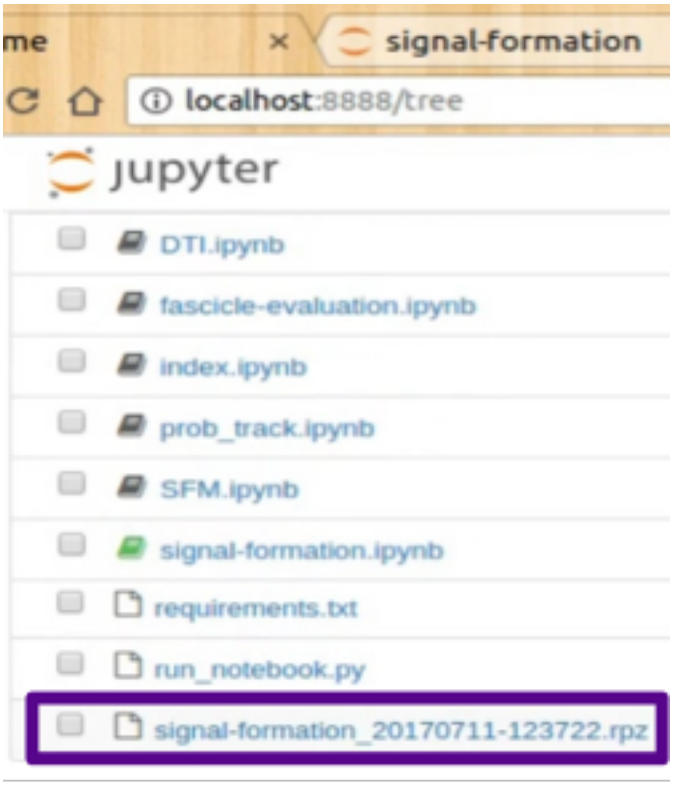
Once you have a notebook that executes the way you want, you can trace and pack all the dependencies, data, and provenance with *reprozip-jupyter* by simply clicking the button on the notebook's toolbar:



The notebook will execute from top-to-bottom and *reprozip-jupyter* traces that execution. If there are no errors in the execution, you'll see two pop-ups like this one after the other:



reprozip-jupyter will name the resulting ReproZip package (*.rpz*) as `notebookname_datetime.rpz` and save it to the same working directory the notebook is in:



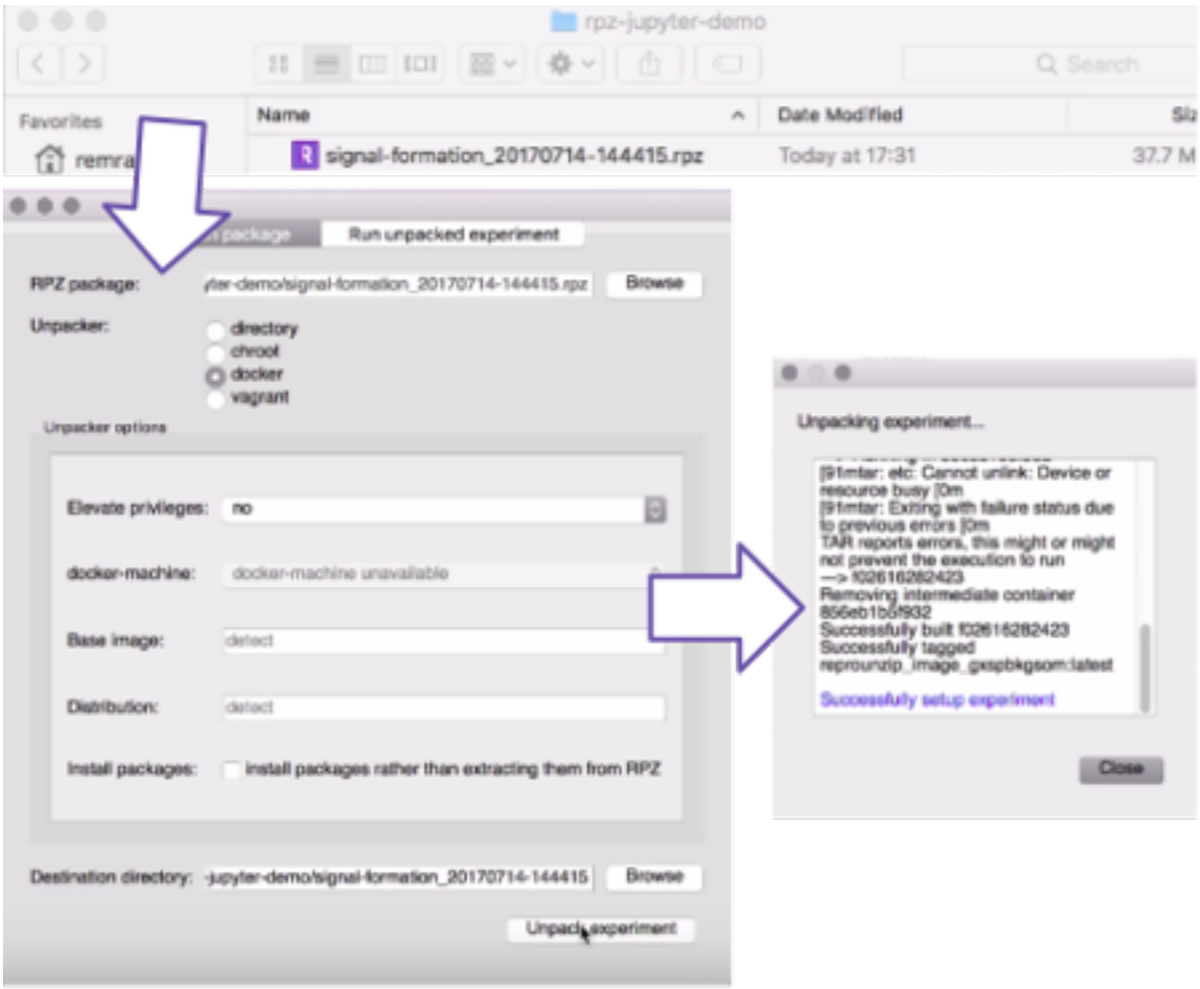
Note that the notebook file itself (`.ipynb`) is not included in the package, so you should share or archive both of those files. The reason is that a lot of services can render notebooks (GitHub, OSF...), and they wouldn't be able to if it was in the RPZ file.

1.6.3 Unpacking

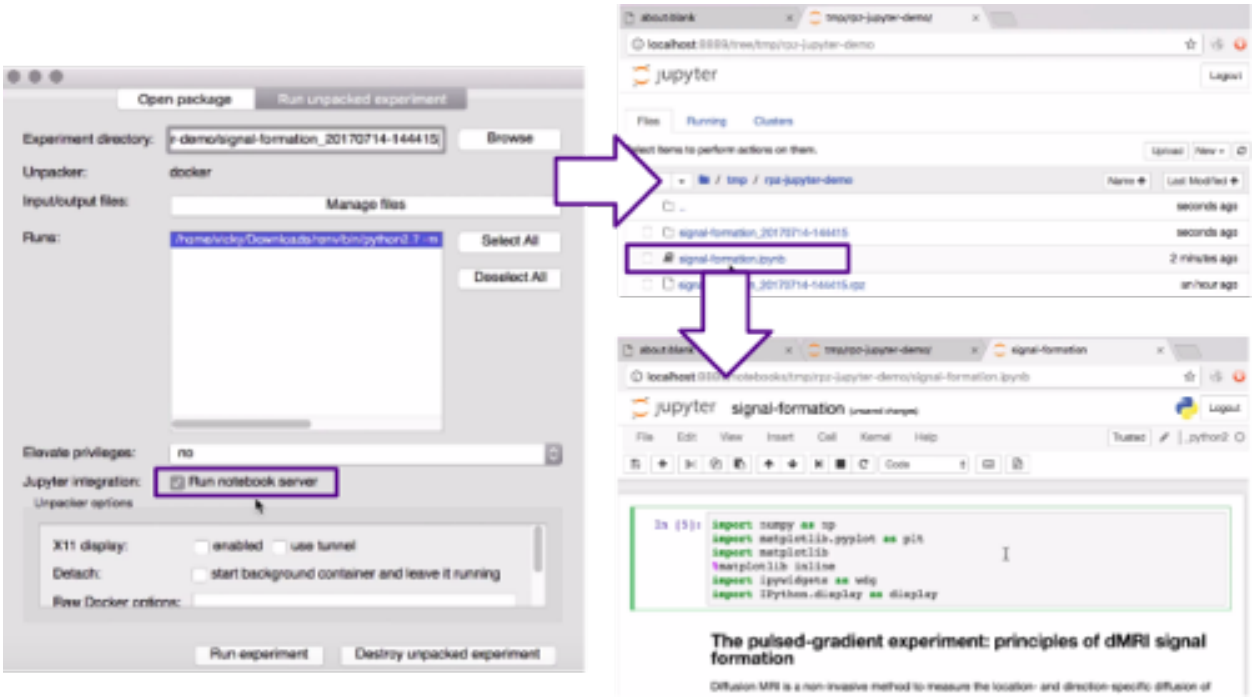
Now, anyone can rerun the Jupyter notebook, with all dependencies automatically configured. First, they would need to install *reprounzip* and the *reprounzip-docker* plugin (see the *installation steps*). Second, they need to download or otherwise acquire the `.rpz` file and original `.ipynb` notebook they'd like to reproduce.

To reproduce the notebook using the GUI, follow these steps:

1. Double-click the .rpz file.
2. The first tab in the window that appears is for you to set up how you'd like ReproUnzip to unpack and configure the contents of the *.rpz*. Choose docker as your unpacker, and choose the directory you'd like to unpack into.
3. Make sure the Jupyter Integration is checked, and click Run experiment:



4. This second table allows you to interact with and rerun the notebook. All you need to do is click 'Run Experiment' and the Jupyter Notebook home file list should pop up in your default browser (if not, navigate to `localhost:8888`). Open the notebook, and rerun with every dependency configured for you!



On the command line, you would:

1. Set up the experiment using *reprounzip-docker*:

```
$ reprounzip docker setup <package.rpz> <directory>
```

2. Rerun the notebook using *reprozip-jupyter*:

```
$ reprozip-jupyter run <directory>
```

3. The Jupyter Notebook home file list should pop up in your default browser (if not, navigate to `localhost:8888`).
4. Open the notebook, and rerun with every dependency configured for you!

1.7 ReproUnzip GUI

reprounzip-qt is a graphical interface (GUI) for reprounzip, allowing you to unpack and reproduce experiments from `.rpz` files without having to use the command-line. You can also set it as the default handler for the `.rpz` file extension so you can open them via double-click.

1.7.1 Installation

reprounzip-qt comes with the installer on Windows and Mac. If you used one of these, you will be able to double click on any `.rpz` file to boot up the GUI.

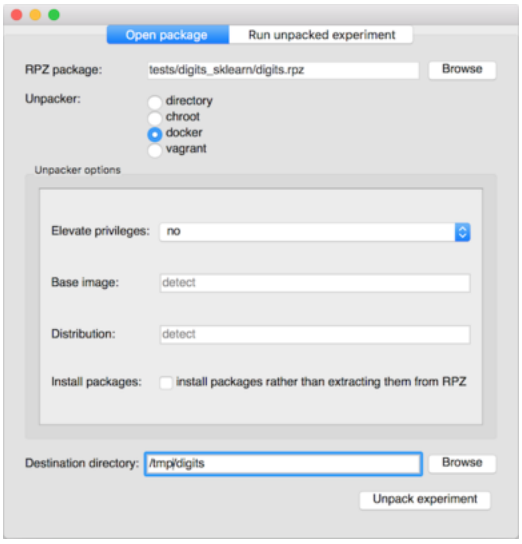
If you are using Anaconda, you can install *reprounzip-qt* from anaconda.org:

```
$ conda install --channel vida-nyu reprounzip-qt
```

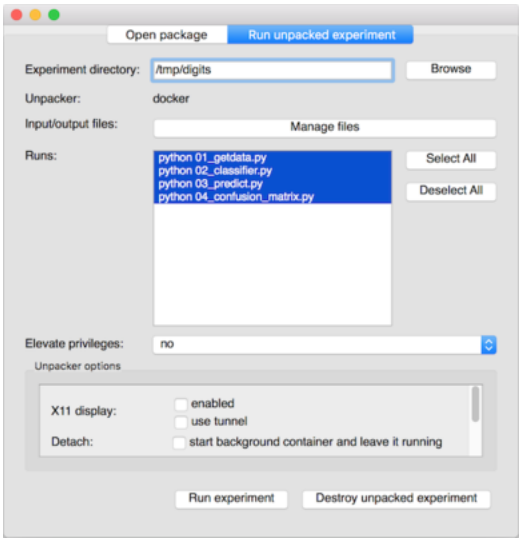
Otherwise, you will need to install PyQt4 (or PyQt5) before you can install *reprounzip-qt* from pip (on Debian or Ubuntu, you can use `apt-get install python-qt4`).

1.7.2 Usage

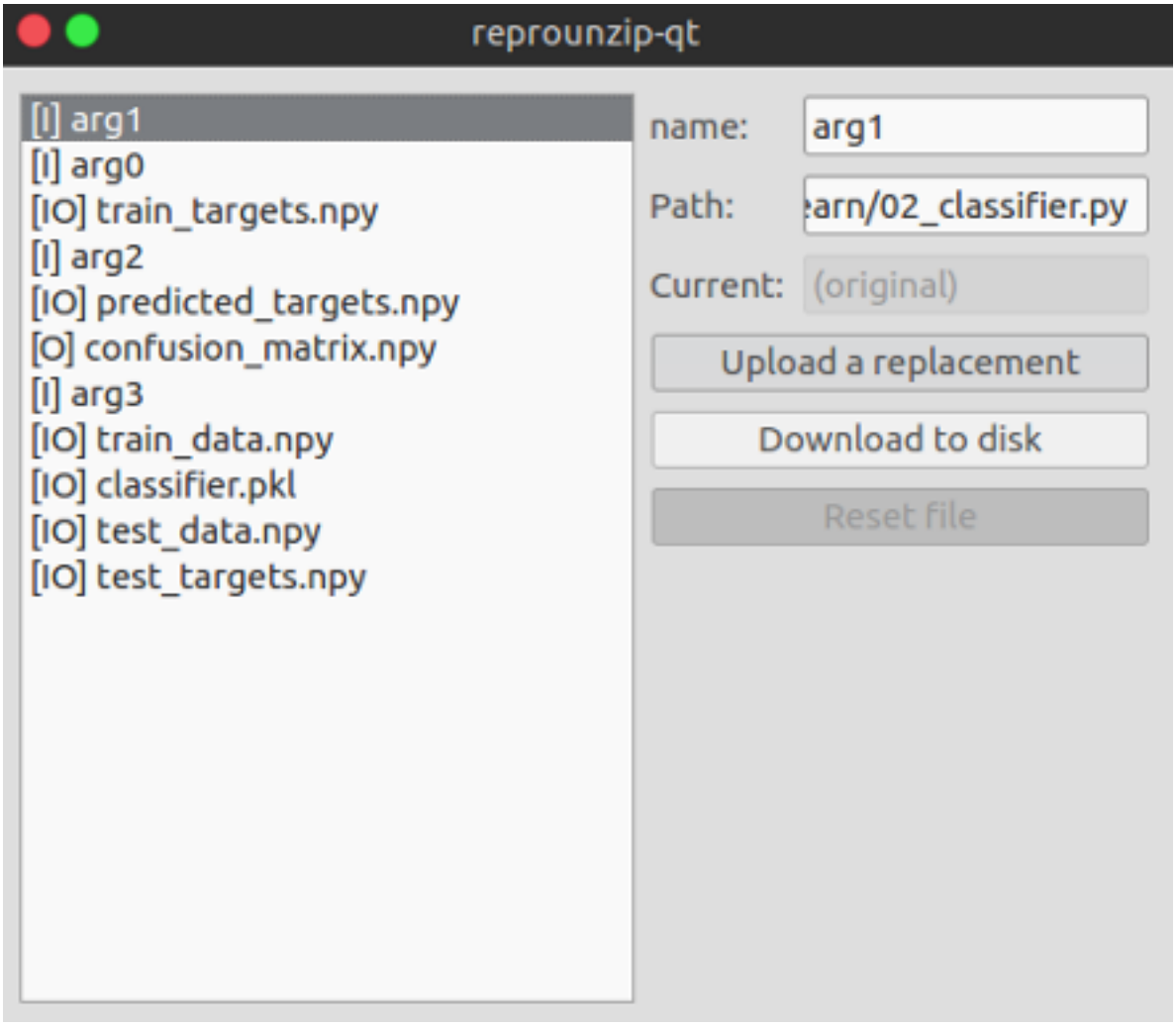
The first tab in the window that appears is for you to set up the experiment. This will allow you to choose which unpacker you'd like to use to reproduce the experiment, and in which directory you'd like to unpack it.



After successfully unpacking, you'll be prompted to run the experiment in the second tab. You can choose which run you want to execute, though the default is to have all runs selected. ReproUnzip will detect the order of the runs and reproduce the experiment accordingly.



Clicking "Manage Files" will bring up options to download the input and output data of the original experiment, and upload your own data to use it in the same experiment. You'll notice input files are marked `[I]` and output files are marked `[O]`. `[IO]` are both input and output files.



1.8 VisTrails Plugin

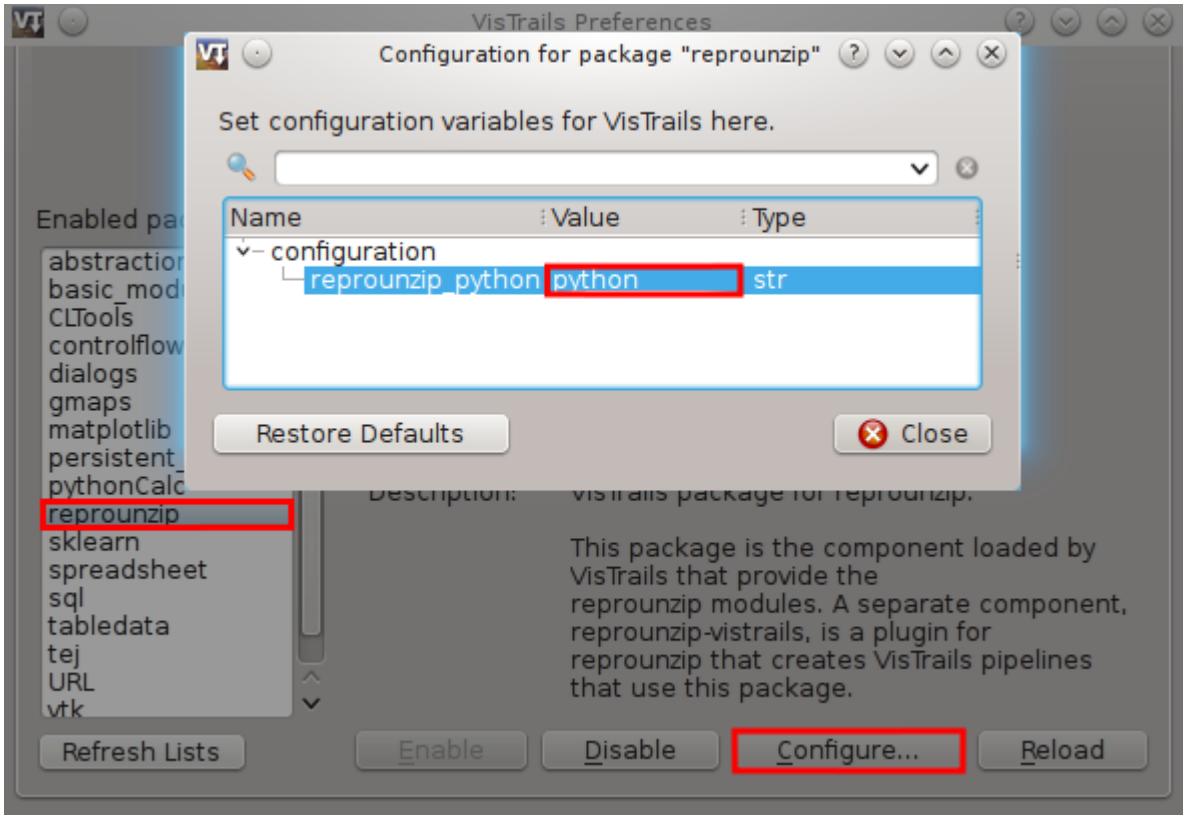
The *reprounzip-vistrails* plugin is a component that interacts with the existing unpackers to generate and execute a VisTrails workflow from the packed experiment. By using VisTrails, you can better manage the experiment workflow: it allows you to run unpacked ReproZip experiments, replace input files, visualize and retrieve output files, and modify the dataflow to re-use steps of the original experiment. For more information about VisTrails, please see their user's guide.

Note: This plugin is **not** distributed with *reprounzip*; it is a separate component that should be installed beforehand (see *Installation* for more details).

Once the plugin is installed, a VisTrails workflow will be generated every time you unpack an experiment; note that this process does not require VisTrails. The workflow file is named `vistrails.vt` and is generated under the unpacked directory.

1.8.1 VisTrails Setup

To run the workflow, you need VisTrails installed on your machine and the reprounzip package, which is included in VisTrails 2.2.3 and up. If you used an installer for either VisTrails or *reprounzip*, you need to set the path to reprounzip's Python interpreter in VisTrails's package configuration dialog:



For example, this will be `/opt/reprounzip/python27/bin/python` if you used the Mac OS X installer, and something similar to `C:\Program Files (x86)\ReproUnzip\python2.7\python.exe` if you used the Windows installer.

1.8.2 Usage

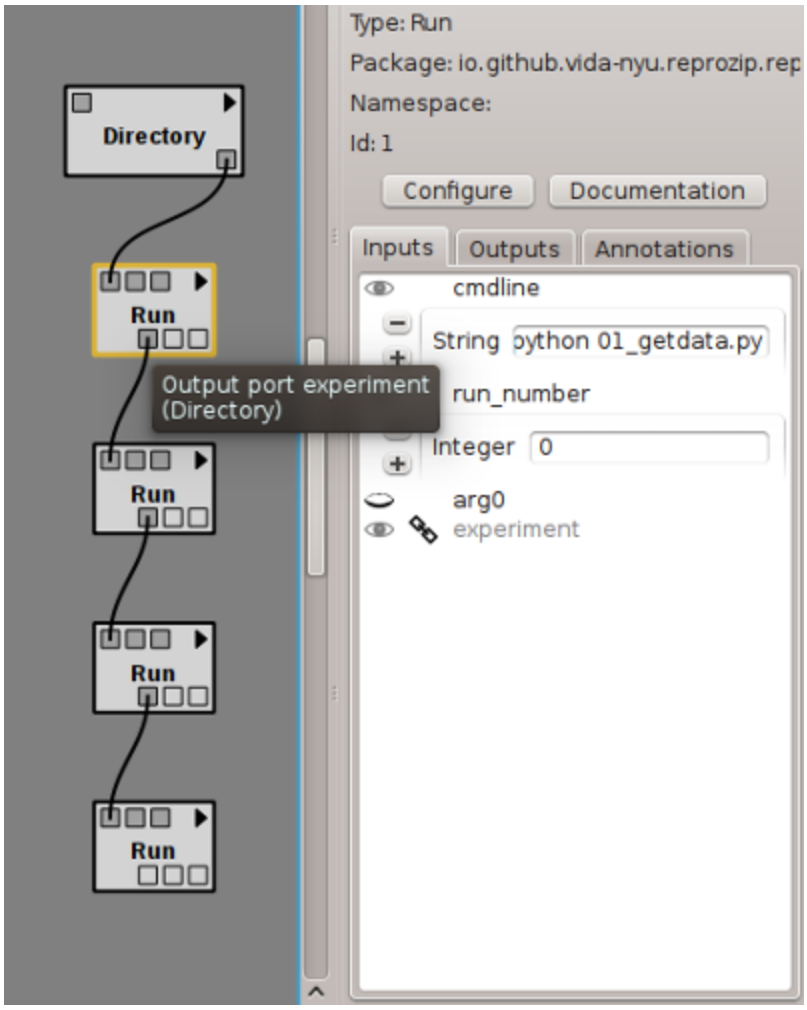
The workflow generated by *reprounzip-vistrails* contains a reference to the unpacked directory and VisTrails modules calling each run in a dataflow; running the workflow is thus the same as running the experiment through *reprounzip run*, except that VisTrails provides caching.

You can open VisTrails and then open the workflow file, which is auto-generated when you unpack an experiment.

The `Directory` module refers to the experiment, while the `Run` module refers to an experiment run. `Directory` is passed from module to module to represent the changes in the environment, since each `Run` will change the internal state of the machine. Note that, if you send the workflow file to another machine, the workflow needs to be updated with the correct path to the unpacked experiment by editing the input port of the `Directory` module.

In a `Run` module, the ports (except the `Directory` one) represent the input and output files that are used by the corresponding run. The module also exposes the command line, should you want to change a parameter or tweak flags.

Note that a file exposed as an output port in one `Run` module may be the input port of the next `Run` module, and yet these are not connected. The dataflow, however, still works since the entire machine state is carried to the next


Directory
Run
Run
Run
Run
Output port experiment
(Directory)
Type: Run
Package: io.github.vida-nyu.reprozip.rep
Namespace:
Id: 1
Configure
Documentation
Inputs
Outputs
Annotations
cmdline
String
python 01_getdata.py
run_number
Integer
0
arg0
experiment

execution. Connecting these ports would work, but would also make *reprounzip* download the file to VisTrails and then upload it again in the same location. You can speed up the workflow by not connecting the files that you do not want to examine or change, since downloading and uploading may take time.

You are encouraged to go through the VisTrails documentation to get familiar with the system.

1.9 Frequently Asked Questions

1.9.1 Can ReproZip pack a client-server scenario?

Yes! However, note that only tracing the client will not capture the full story: reproducibility is better achieved (and guaranteed) if the server is traced as well. Having said that, currently, ReproZip can only trace local servers: if the server is remote (i.e., running in another machine), ReproZip cannot capture it. In this case, you can trace the client, and the experiment can only be reproduced if the remote server is still running at the moment of the reproduction.

The easiest way to pack a local client-server experiment is to write a script that starts the server, runs your client(s), and then shuts down the server; ReproZip can then trace this script. See *Capturing Connections to Servers* for more information.

1.9.2 Can ReproZip pack a database?

ReproZip can trace a database server; however, because of the format it uses to store data (and also because different databases work differently), it might be hard for you to control exactly what data will be packed. You probably want to pack all the data from the databases/tables that your experiment uses and not just the pages that were touched while tracing the execution. This can be done by inspecting the configuration file and adding the relevant patterns that cover the data, e.g.: for MySQL:

```
additional_patterns:
  - /var/lib/mysql/**
```

See *Capturing Connections to Servers* for an example using a local database and additional information.

Note that ReproZip does not currently save the state of the files. Therefore, if your experiment modifies a database, ReproZip will pack the already modified data (not the one before tracing the experiment execution).

1.9.3 Can ReproZip pack interactive tools?

Yes! The *reprounzip* component should have no problems with experiments that interact with the user through the terminal. If your experiment runs until it receives a Ctrl+C signal, that is fine as well: ReproZip will not interfere unless you press Ctrl+C twice, stopping the experiment.

GUI tools are also supported; see *Can ReproZip pack graphical tools?* for more information.

1.9.4 Can ReproZip pack graphical tools?

Yes! On Linux, graphical display is handled by the X server. Applications can connect to it as clients to display their windows and components, and to get user input. Most unpackers now support forwarding the X connection from the experiment to the X server running on the unpacking machine. You will need a running X server to make this work, such as Xming for Windows or XQuartz for Mac OS X. If you are running Linux, chances are that an X server is already configured and running.

X support is **not** enabled by default; to enable it, use the flag `--enable-x11` in the `run` command of your preferred unpacker.

> **Warning:** While displaying a UI through the X protocol works fine, applications using direct rendering (DRI) to access dedicated graphic hardware might not be reproducible: the libGL library packed with the experiment is often specific to the driver of the original machine, and therefore the machine where the experiment is being reproduced would need to use the exact same hardware and driver.
>
> Please refrain from requiring direct rendering in applications that you intend to pack with ReproZip.

If using Vagrant, you can also use the virtual machine's native display directly, by supplying the `--use-gui` option to `reprounzip vagrant setup`.

1.9.5 How can I access the generated system or virtual machine directly?

If you are running `reprounzip vagrant`, you can connect to the Vagrant virtual machine by running `vagrant ssh` or connecting via SSH to the destination listed by `vagrant ssh-config` (often `localhost:2222`). These commands should be run from inside the directory created by the unpacker.

If you are running `reprounzip docker`, you can inspect the Docker container directly by using `docker`, or start a new one based on the image created by *reprounzip*; all images generated by ReproZip are named with the `reprounzip_` prefix. For more information on how to inspect and create Docker containers, please refer to the Docker documentation.

For `reprounzip chroot` and `reprounzip directory`, the filesystem is in the `root` subdirectory under the experiment path.

See *Structure of Unpacked Experiments* for more details.

> **Warning:** Note that, in the generated system, only the files needed for running the unpacked experiment are guaranteed to work correctly. This means that you may have only parts of a software distribution (required to run the experiment), but not the software in its entirety (unless the complete software was included in the configuration file while packing). For example, you may only have a few Python files that the experiment needs, but not the ones required to run Python interactively or install new libraries. Therefore, do not expect that all the software components will run smoothly when acessing the system.
>
> The utilities from the base system might also not work correctly (if they are not part of the experiment) because *reprounzip* overwrites the libraries with the ones from the original environment. In the worst-case scenario, the dynamic linker or the shell may not be usable. Note that some unpackers install `/bin/busybox`, which you may find helpful.

1.9.6 What if my experiment runs on a distributed environment?

ReproZip cannot trace across multiple machines. You could trace each component separately, but ReproZip cannot connect these multiple `.rpz` files to setup the multiple machines the right way. In particular, you will probably need to set up the same network for the components to talk to each other.

1.10 Troubleshooting

The best way to start solving an issue in ReproZip is probably to look at the log messages. Some messages are not displayed by default when running ReproZip, but you can use the `--verbose` (or `-v`) flag to display them. In addition, all the log messages are stored under `$HOME/.reprozip/log`.

Please feel free to contact us at users@reprozip.org if you encounter issues while using ReproZip.

Issue " *reprozip* **does not identify my input/output file."**

Diagnosis ReproZip uses some heuristics to identify an input or output file. However, this is only intended to be a starting point, since these heuristics may fail.

Solution You should check the configuration file and edit the `inputs_outputs` section if necessary; giving readable names to input/output files also helps during reproduction. Please refer to *Editing the Configuration File* for more information.

Issue "None of my files are packed when tracing a daemon."

Diagnosis If you are starting the daemon via the *service* tool, it might be calling *init* over a client/server connection. In this situation, ReproZip will successfully pack the client, but anything the server (*init*) does will not be captured.

Solution You can still trace the binary or a non-systemd *init* script directly. For example, instead of:

```
$ reprozip trace service mysql start
```

you can trace either the *init* script:

```
$ reprozip trace /etc/init.d/mysql start
```

or the binary:

```
$ reprozip trace /usr/bin/mysqld
```

Note that, if you choose to trace the binary, you need to figure out the right command line options to use. Also, make sure that systemd is not called, since ReproZip and systemd currently do not get along well.

Issue " *reprozip* **fails with** `couldn't use ptrace` **"**

Diagnosis `ptrace` is the mechanism that ReproZip uses to attach to another process and follow its system calls. Because it is so powerful, some security policies, environments or isolation mechanism may disable it.

Solution

- If you are using Docker, you can use the Docker option `--cap-add=SYS_PTRACE` (or provide your own seccomp profile that allows ptrace, by adding `"ptrace"` to the default profile; see the Docker documentation on seccomp).

Issue " *reprounzip* **cannot get an output file using** `download` **after reproducing the experiment."**

Diagnosis This is probably the case where this output file does not have a fixed path name. It is common for experiments to dynamically choose where the outputs should be written, e.g.: by putting the date and time in the filename. However, ReproZip uses filenames in the `inputs_outputs` section of the configuration file to detect those when reproducing the experiment: if the name of the output file when reproducing is different from when it was originally packed, ReproZip cannot detect these as output files, and therefore, cannot get them through the `download` command.

Solution The easiest way to solve this issue is to re-pack the experiment: write a simple bash script that runs the experiment and either renames outputs or creates symbolic links to them with known filenames; then, trace this script (instead of the actual entry-point of your experiment) and specify these fixed path names in the `inputs_outputs` section of the configuration file.

Issue " *reprounzip-vagrant* **installation fails with error** `Unable to find vcvarsall.bat` **on Windows."**

Diagnosis Python is trying to build PyCrypto, one of the dependencies of *reprounzip-vagrant*, but there is no C compiler available.

Solution You can either build PyCrypto from source, or follow the instructions on this website to get the non-official binaries.

Issue " *reprounzip-vagrant* **installation fails with error** `unknown argument: '-mno-fused-madd'` **on Mac OS X."**

Diagnosis This is an issue with the Apple LLVM compiler, which treats unrecognized command-line options as errors.

Solution As a workaround, before installing *reprounzip-vagrant*, run the following:

```
$ export CFLAGS="-Wno-error=unused-command-line-argument-hard-error-in-
↪future"
```

Then, re-install *reprounzip-vagrant*:

```
$ pip install -I reprounzip-vagrant
```

Or use the following command in case you want all the available plugins:

```
$ pip install -I reprounzip[all]
```

Issue "The experiment fails with `Error: Can't open display: :0` **when trying to reproduce it."**

Diagnosis The experiment probably involves running a GUI tool.

Solution The *reprounzip* component supports GUI tools, but it is not enabled by default; add the flag `--enable-x11` to the `run` command to enable it. See *Can ReproZip pack graphical tools?* for more information.

Issue "The experiment run with *reprounzip directory* **fails to find a file that has been packed."**

Diagnosis The *directory* unpacker does not provide any isolation from the filesystem: if the experiment being reproduced use absolute paths, these will point outside the experiment directory, and files may not be found.

Solution Make sure that the experiment does not use any absolute paths: if only relative paths are used internally and in the command line, `reprounzip directory` should work. As an alternative, you can use other unpackers (e.g.: `reprounzip chroot` and `reprounzip vagrant`) that work in the presence of hardcoded absolute paths.

Issue " *reprounzip* **fails with** `DistributionNotFound` **errors."**

Diagnosis You probably have some plugins left over from a previous installation.

Solution Be sure to upgrade or remove outdated plugins when you upgrade *reprounzip*. The following command may help:

```
$ pip install -U reprounzip[all]
```

Issue " *reprounzip* **shows** `running in chroot, ignoring request` **."**

Diagnosis This message comes from the systemd client, which will probably not work with ReproZip.

Solution In this case, the experiment should be re-packed without using systemd (see *this issue* for more information).

Issue " `reprounzip vagrant setup` **fails to resolve a host address."**

Diagnosis When running `reprounzip vagrant setup`, if you get an error similar to this:

```
==> default: failed: Temporary failure in name resolution.
==> default: wget: unable to resolve host address ...
```

there is probably a firewall blocking the Vagrant VM to have Internet connection; the VM needs Internet connection to download required software for setting up the experiment for you.

Solution Make sure that your anti-virus/firewall is not causing this issue.

Issue "The experiment fails because of insufficient memory in Vagrant."

Diagnosis It is possible that the default amount of memory allocated to the VM is insufficient for the experiment. You can see a lot of different messages there, including:

- `Out of memory`
- `Could not allocate memory`
- `Killed`

Solution From VirtualBox, stop the machine and allocate more memory under *Settings* > *System* > *Motherboard* > *Memory*.

You can also use the `--memory` option when you run `reprounzip vagrant setup` to specify the amount of memory (in megabytes) at that time.

Issue " `reprounzip run` **fails with** `no such file or directory` **or similar."**

Diagnosis This error message may have different reasons, but it often means that a specific version of a library or a dynamic linker is missing:

1. If you are requesting *reprounzip* to install software using the package manager (by running `reprounzip installpkgs`), it is possible that the software packages from the package manager are not compatible with the ones required by the experiment.
2. If, while packing, the user chose not to include some packages, *reprounzip* will try to install the ones from the package manager, which may not be compatible.
3. If you are using `reprounzip vagrant` or `reprounzip docker`, ReproZip may be failing to detect the closest base system for unpacking the experiment.

Solution

1. Use the files inside the experiment package to ensure compatibility.
2. Contact the author of the ReproZip package to ask for a new package with all software packages included.
3. Try a different base system that you think it is closer to the original one by using the option `--base-image` when running these unpackers.

Issue "There are warnings from requests/urllib3 when running ReproZip."

```
/usr/local/lib/python2.7/dist-packages/requests/packages/urllib3/util/
↪ssl_.py:79:
InsecurePlatformWarning: A true SSLContext object is not available. This
prevents urllib3 from configuring SSL appropriately and may cause␣
↪certain SSL
connections to fail. For more information, see
https://urllib3.readthedocs.io/en/latest/security.html
↪#insecureplatformwarning.
```

Diagnosis Most Python versions are insecure, because they do not validate SSL certificates, thus generating these warnings.

Solution If you are using Python 2.7.9 and later, you shouldn't be affected, but if you see `InsecurePlatformWarning`, you can run `pip install requests[security]`, which should bring in the missing components.

1.11 Structure of Unpacked Experiments

While *reprounzip* is designed to allow users to reproduce an experiment without having to master the tool used to run it (e.g.: Vagrant and Docker), in some situations it might be useful to go behind the scenes and interact with the unpacked experiments directly.

This page describes in more details how the unpackers operate.

Note: Future versions of unpackers might work in a different way. No attempt is made to make unpacked experiments compatible across different versions of *reprounzip*. Packages will always be compatible though.

1.11.1 Common Files across Unpackers

The unpacked directory contains the original configuration file as `config.yml`. In fact, the VisTrails integration relies on it.

A file named `.reprounzip` also marks the directory as an unpacked experiment. This is a Python pickle file containing a dictionary with various types of information:

- `unpacker` maps to the unpacker's name.
- `input_files` is used by the uploader/downloader machinery to keep the state of the input files inside the experiment, as they may be replaced by the user or overwritten by runs.
- Other information specific to the unpacker, as described next.

1.11.2 The *directory* Unpacker

The experiment directory contains:

- The original configuration file `config.yml`.
- The pickle file `.reprounzip`.
- The tarball `inputs.tar.gz`, which contains the original files that were identifies as input files. This tarball is used for file restoration using `upload :<input-id>` (see *Managing Input and Output Files*).
- A directory called `root`, which contains all the packaged files in their original path, with symbolic links to absolute paths rewritten to prepend the path to `root`.

```
unpacked-directory/
    .reprounzip
    config.yml
    inputs.tar.gz
    root/
        ...
```

When running the `run` command, the unpacker sets `LD_LIBRARY_PATH` and `PATH` to point inside `root`, and optionally `DISPLAY` and `XAUTHORITY` to the host's ones.

1.11.3 The *chroot* Unpacker

The experiment directory contains:

- The original configuration file `config.yml`.
- The pickle file `.reprounzip`, which stores whether magic directories are mounted, as explained below.
- The tarball `inputs.tar.gz`, which contains the original files that were identifies as input files. This tarball is used for file restoration using `upload :<input-id>` (see *Managing Input and Output Files*).
- A directory called `root`, which contains all the packaged files in their original path, with no symbolic links rewritten and file ownership restored.

```
unpacked-directory/
    .reprounzip
    config.yml
    inputs.tar.gz
    root/
        dev/
        dev/pts/
        proc/
        ...
```

If a file is listed in the configuration file but wasn't packed (i.e.: `pack_files` was set to `false` for a software package), such file is copied from the host; if this file does not exist on the host, a warning is shown when unpacking.

Unless `--dont-bind-magic-dirs` is specified when unpacking, the special directories `/dev`, `/dev/pts`, and `/proc` are mounted with `mount -o bind` from the host. Also, if `/bin/sh` or `/usr/bin/env` weren't both packed, a static build of busybox is downloaded and put under `/bin/busybox`, and the missing binaries are created as symbolic links pointing to busybox.

Should you require a shell inside the experiment environment, you can use:

```
chroot root/ /bin/sh
```

1.11.4 The *vagrant* Unpacker

The experiment directory contains:

- The original configuration file `config.yml`.
- The pickle file `.reprounzip`, which stores whether a chroot is used, as explained below.
- The tarball `data.tgz`, which is part of the `.rpz` file and used to populate the virtual machine (VM) when it gets created.
- The setup script `setup.sh`.
- The file `rpz-files.list`, which contains the list of files to unpack. This list is passed to `tar -T` while unpacking.
- A `Vagrantfile`, which is used to build the VM.

```
unpacked-directory/
    .reprounzip
    config.yml
    data.tgz
    busybox
    Vagrantfile
    setup.sh
    rpz-files.list
```

Once `vagrant up` has been run by the `setup/start` command, a `.vagrant` subdirectory is created, and its content is managed by Vagrant (and appears to vary among different platforms).

Note that Vagrant drives VirtualBox or a similar virtualization software to run the VM. These will maintain state outside of the experiment directory. If you need to reconfigure or otherwise interact with the VM, you should do it from that virtualization software (e.g.: VirtualBox). The VM is named as the experiment directory with an additional suffix.

There are two modes for the virtual machine, controlled through command-line flags:

- The default mode, `--use-chroot`, creates a chroot environment inside the VM at `/experimentroot`. This allows ReproZip to unpack very different file system hierarchies without breaking the base system of the VM (in particular, `ssh` needs to keep working for the VM to be usable). In this mode, software packages that were not packed (i.e.: `pack_files` was set to `false`) are installed in the VM and their required files are copied to the `/experimentroot` hierarchy. The software packages that were packed are simply copied over without any interaction with the VM's system.
- If `--dont-use-chroot` is used, no chroot environment is created. Files from software packages are never copied from the `.rpz` file; instead, they get installed from the package manager. Other files are simply unpacked in the VM system, possibly overwriting existing files. As long as *reprounzip-vagrant* manages to find a VM image with the same operating system as the original one, reproduction is expected to work reliably.

In the `--use-chroot` mode, a static build of busybox is downloaded and put under `/experimentroot/busybox`, and if `/bin/sh` wasn't packed, it is created as a symbolic link pointing to busybox.

Uploading and downloading files from the environment is done via the shared directory `/vagrant`, which is the experiment directory mounted in the VM by Vagrant.

Should you require a shell inside the experiment environment, you can use:

```
vagrant ssh
```

Please be aware of whether `--use-chroot` is in use when accessing the experiment environment: in this case, the experiment's files are located under `/experimentroot`.

1.11.5 The *docker* Unpacker

The experiment directory contains:

- The original configuration file `config.yml`.
- The pickle file `.reprounzip`, which stores the name of the images built by the unpacker, as explained below.
- The tarball `data.tgz`, which is part of the `.rpz` file and used to populate the Docker container.
- The file `rpz-files.list`, which contains the list of files to unpack. This list is passed to `tar -T` while unpacking.
- A `Dockerfile`, which is used to build the original image.

```
unpacked-directory/
    .reprounzip
    config.yml
    data.tgz
    busybox
    rpzsudo
    Dockerfile
    rpz-files.list
```

Static builds of busybox and rpzsudo are always downloaded and put into the Docker image as `/busybox` and `/rpzsudo`, respectively.

Note that the `docker` command connects to a Docker daemon over a socket and that state will be changed there. The daemon might not be local; in particular, `docker-machine` might be used, which allows *reprounzip-docker* to be used on non-Linux machines, and the daemon might be in a virtual machine, on another host, or in the cloud. The *docker* unpacker will keep the environment variables set when calling Docker, notably `DOCKER_HOST`, so these can be set accordingly before running the unpacker.

Images and containers built by the unpacker are given a random name with the prefixes `reprounzip_image_` and `reprounzip_run_`, respectively; they are cleaned up when the `destroy` command is invoked. There are two images of which *reprounzip-docker* keeps track in the `.reprounzip` pickle file: the initial image, i.e., the one built by `setup/build` by calling `docker build`, and the current image (initially the same as the initial image), which has been affected by a number of `run` and `upload` calls. Running the `reset` command returns to the initial image without having to rebuild. After each `run` invocation, the container is committed to a new current image so that state is kept.

A `--detach` option allows to start container and forget about them. reprounzip-docker leaves the container running and doesn't wait for it; this means that you can start a service on a remote machine, but note that because that container won't be committed to a new image, the side-effects of running it won't affect later executions on the same unpacked folder.

Uploading files to the environment is done by running a simple Dockerfile that builds a new image. Downloading files is done via the `docker cp` command.

1.12 Developer's Guide

1.12.1 General Development Information

Development happens on GitHub; bug reports and feature requests are welcome. If you are interested in giving us a hand, please do not hesitate to submit a pull request there.

Continuous testing is provided by Travis CI. Note that ReproZip supports both Python 2 and 3. Test coverage is not very high because there are a lot of operations that are difficult to cover on Travis (for instance, Vagrant VMs cannot be used over there).

If you have any questions or need help with the development of an unpacker or plugin, please use our development mailing-list at dev@reprozip.org.

1.12.2 Writing Unpackers

ReproZip is divided into two steps. The first is packing, which gives a generic package containing the trace SQLite database, the YAML configuration file (which lists the paths, packages, and metadata such as command line, environment variables, and input/output files), and actual files. In the second step, a package can be run using *reprounzip*. This decoupling allows the reproducer to select the unpacker of his/her desire, and also means that when a new unpacker is released, users will be able to use it on their old packages.

Currently, different unpackers are maintained: the defaults ones (`directory` and `chroot`), `vagrant` (distributed as reprounzip-vagrant) and `docker` (distributed as reprounzip-docker). However, the interface is such that new unpackers can be easily added. While taking a look at the "official" unpackers' source is probably a good idea, this page gives some useful information about how they work.

ReproZip Pack Format (`.rpz`)

An `.rpz` file is a `tar.gz` archive that contains two directories: `METADATA`, which contains meta-information from *reprozip*, and `DATA`, which contains the actual files that were packed and that will be unpacked to the target directory for reproducing the experiment.

The `METADATA/config.yml` file is in the same format as the configuration file generated by *reprozip*, but without the `additional_patterns` section (at this point, it has already been expanded to the actual list of files while packing).

The `METADATA/trace.sqlite3` file is the original trace generated by the C tracer and maintained in a SQLite database; it contains all the information about the experiment, in case the configuration file is insufficient in some aspect. This file is used, for instance, by the *graph* unpacker, so that it can recover the exact hierarchy of processes, together with the executable images they execute and the files they access (with the time and mode of these accesses).

See also:

Trace Database Schema

Structure

An unpacker is a Python module. It can be distributed separately or be a part of a bigger distribution, given that it is declared in that distribution's `setup.py` as an *entry_point* to be registered with *pkg_resources* (see setuptools'

dynamic discovery of services and plugins section). You should declare a function as *entry_point* `reprounzip.unpackers`. The name of the entry_point (before =) will be the *reprounzip* subcommand, and the value is a callable that will get called with the `argparse.ArgumentParser` object for that subcommand.

The package `reprounzip.unpackers` is a namespace package, so you should be able to add your own unpackers there if you want to. Please remember to put the correct code in the `__init__.py` file (which you can copy from here) so that namespace packages work correctly.

The modules `reprounzip.common`, `reprounzip.utils`, and `reprounzip.unpackers.common` contain utilities that you might want to use (make sure to list *reprounzip* as a requirement in your `setup.py`).

Example of `setup.py`:

```
setup(name='reprounzip-vagrant',
      namespace_packages=['reprounzip', 'reprounzip.unpackers'],
      install_requires=['reprounzip>=0.4'],
      entry_points={
          'reprounzip.unpackers': [
              'vagrant = reprounzip.unpackers.vagrant:setup'
              # The setup() function sets up the parser for reprounzip vagrant
          ]
      }
      # ...
)
```

Usual Commands

If possible, you should try to follow the same command names that the official unpackers use, which are:

- `setup`: to create the experiment directory and set everything for execution;
- `run`: to reproduce the experiment;
- `destroy`: to bring down all that setup and to prepare and delete the experiment directory safely;
- `upload` and `download`: to replace input files in the experiment, and to get the output files for further examination, respectively.

If these commands can be broken down into different steps that you want to expose to the user, or if you provide completely different actions from these defaults, you can add them to the parser as well. For instance, the *vagrant* unpacker exposes `setup/start`, which starts or resumes the virtual machine, and `destroy/vm`, which stops and deallocates the virtual machine but leaves the template for possible reuse.

A Note on File Paths

ReproZip supports Python 2 and 3, is portable to different operating systems, and is meant to accept a wide variety of configurations so that it is compatible with most experiments out there. Even trickier, *reprounzip-vagrant* needs to manipulate POSIX filenames on Windows, e.g.: in the unpacker. Therefore, the rpaths library is used everywhere internally. You should make sure to use the correct type of path (either `PosixPath` or `Path`) and to cast these to the type that Python functions expect, keeping in mind 2/3 differences (most certainly either `filename.path` or `str(filename)`).

Experiment Directory Format

Unpackers usually create a directory with everything necessary to later run the experiment. This directory is created by the `setup` operation, cleaned up by `destroy`, and is the argument to every command. For example, with *reprounzip-vagrant*:

```
$ reprounzip vagrant setup someexperiment.rpz mydirectory
$ reprounzip vagrant upload mydirectory /tmp/replace.txt:input_text
```

Unpackers unpack the config.yml file to the root of that directory, and keep status information in a `.reprounzip` file, which is a dict in `pickle` format. Following the same structure will allow the `showfiles` command, as well as `FileUploader` and `FileDownloader` classes, to work correctly. Please try to follow this structure.

Signals

Since version 0.4.1, *reprounzip* has signals that can be used to hook in plugins, although no such plugin has been released at this time. To ensure that these work correctly when using your unpacker, you should emit them when appropriate. The complete list of signals is available in signal.py.

1.12.3 Final Observations

After reading this page, reading the source code of one of the "official" unpackers is probably the best way of understanding how to write your own. They should be short enough to be easy to grasp. Should you have additional questions, do not hesitate to use our development mailing-list: *dev@reprozip.org*.

1.13 Glossary

configuration (file) A YAML file generated by `reprozip trace` and read by `reprozip pack`. It can be edited before creating the package to control which files are to be included. It also contain other metadata used during reproduction. See *Editing the Configuration File*.

distribution package A software component installed by the Linux distribution's package manager. ReproZip tries to identify from which distribution package each file comes; this allows the reproducer to install the software from his distribution's package manager instead of extracting the files from the `.rpz` file.

package (or pack) A `.rpz` file generated by `reprozip pack`, containing all the files and metadata required to reproduce the experiment on another machine. See *Using reprozip*.

run A single command line traced by `reprozip trace [--continue]`. Multiple commands can be traced successively before creating the pack; the reproducer will be able to run them separately using `reprounzip <unpacker> run <directory> <run-id>`.

software package The same as a distribution package.

unpacker A plugin for the *reprounzip* component that reproduces an experiment from a `.rpz` package. The unpackers *chroot*, *directory*, and *installpkgs* are distributed with *reprounzip*; others come in separate packages (*reprounzip-docker* and *reprounzip-vagrant*). See *Unpackers*.

CHAPTER 2

Links

- Project website
- GitHub repository
- Mailing lists:
  - users@reprozip.org (users)
  - dev@reprozip.org (developers)

Index