ReproZip Documentation

Release 0.4

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

September 30, 2014

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* actually 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 substitute 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 also to build on them, alowing 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.

1.2 Installation

ReproZip is available as open source, released under the Revised BSD License. Please visit ReproZip's website to find links to our PyPI packages or our GitHub repository.

1.2.1 Software Requirements

ReproZip is comprised of two components: **reprozip** (for the packing step) and **reprounzip** (for the unpack step). Additional plugins are also provided for *reprounzip*: **reprounzip-vagrant**, which unpacks the experiment in a Vagrant virtual machine, and **reprounzip-docker**, which unpacks the experiment in a Docker container (please see *Additional Unpackers* for more information). More plugins may be developed in the future (and of course, you are free to *roll your own*).

These are all standard Python packages that you can install using pip. However, *reprozip* only works on Linux and needs a C compiler recognized by distutils since it includes a C extension module that will be built during installation.

The operating system compatibility for the two ReproZip components is the following:

Component	Linux	Mac OS X	Windows
reprozip	Yes	No	No
reprounzip	Yes	Yes [2]	Yes [1]

Python 2.7.3 or greater [3] is required to run ReproZip. Besides, depending on the component or plugin to be used, some additional software packages are also required, as described below:

Component / Plugin	Required Software Packages
reprozip	SQLite, a working C compiler
reprounzip	None
reprounzip-vagrant	Vagrant
reprounzip-docker	Docker

1.2.2 Obtaining the Software

In ReproZip, the components must be installed separately as they fulfill different purposes (and typically, you will use them on different machines). First, you will need Python and pip. Then, to install a ReproZip component, simply run the following command:

```
$ pip install <name>
```

where <*name*> is the name of the component.

The additional plugins for *reprounzip* can also be installed using the same command. They depend on *reprounzip*, so it will be installed automatically if you simply install the plugin.

1.3 Using *reprozip*

The *reprozip* component is responsible for packing an experiment. In ReproZip, we assume that the experiment can be executed by a single command line, preferably with no GUI involved (please refer to *Further Considerations* for additional information regarding different types of experiments).

There are three steps when packing an experiment with *reprozip*: *tracing the experiment*, *editing the configuration file* if necessary, and *creating the reproducible package for the experiment*. Each of these steps is explained in more details below. 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 an experiment:

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

[1] By using either *reprounzip-vagrant* or *reprounzip-docker*.

[2] By using either *reprounzip-vagrant* or *reprounzip-docker*.

[3] This is because of Python bug 13676 related to sqlite3.

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

By default, if the operating system is Debian or Debian-based (e.g.: Ubuntu), *reprozip* will also try to automatically identify the distribution packages that the files come from, 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, the *trace* command can take some time doing that after the experiment has finished, 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`, created under the path where the *reprozip* command was issued.

1.3.2 Editing the Configuration File

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

It is possible to not change anything, as the automatically-generated file is probably sufficient to generate a working package, however you may want to edit this file prior to the creation of the package in order to add or remove files. This can be particularly useful, for instance, to remove big files that can be obtained elsewhere when reproducing the experiment, so as 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, as well as to add or remove environment variables.

The first part of the configuration file gives general information with respect to the experiment execution, including the command line, environment variables, main input and output files, and machine information:

```
# Run info
version: <reprozip-version>
runs:
- 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>
  input_files: <input-files>
  output_files: <output-files>
  system: <system-kernel>
  uid: <user-id>
  workingdir: <working-directory>
```

If necessary, users may change the command line parameters by editing <*command-line-arguments*>, and add or remove environment variables by editing <*environment-variables*>. Other attributes should mostly not be changed, apart from the *input_files* and *output_files* (in particular, feel free to remove some of these, and give them more descriptive names).

The next section in the configuration file shows the files to be packed. If the software dependencies were identified by the package manager of the system during the *trace* command execution, they will be listed under *packages*; the file dependencies not identified in software packages are 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 which software packages 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*.

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 (contrieved 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
```

1.3.3 Creating a Package

After tracing the experiment and optionally editing the configuration file, the experiment package can be created by issuing the command below:

```
$ 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 this is only at this point that files will be copied from your environment and into the package; as such, you should not change any file that the experiment used before packing it, or else the package will contain different files than the ones the experiment used while it was traced.

1.3.4 Further Considerations

Packing Multiple Command Lines

ReproZip is meant to trace a whole experiment in one go. Therefore, if an experiment comprises multiple successive commands, users should create a simple **script** that runs all these commands, and pass *that* with `reprozip trace`.

Packing GUI and Interactive Tools

Currently, ReproZip cannot ensure that GUI interfaces will be correctly reproduced (support is coming soon), so we recommend packing tools in a non-GUI mode for a successfull reproduction.

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 in that case when reproduced on a different machine.

Capturing Connections to Servers

Communication with remote servers is outside the scope of ReproZip: when reproducing an execution, 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 can 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 whole script, rather than the experiment itself. 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.

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

ReproZip 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 don't move. 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 default name) that always points to the latest result, and use that as the output file's path in the configuration.

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 (see *Unpacking an Experiment in Linux*), but more unpackers can be provided through plugins; some of these are compatible with different environment as well (see *Additional Unpackers*).

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 following command allows users to do so:

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

where *<package>* corresponds to the experiment package (i.e.: the `.rpz` file). You can pass `-v` (for *verbose*) or `-v -v` to get more detailed information on the package.

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 execution:

```
----- 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>)
Executions:
    <command-line>
        wd: <working-directory>
        exitcode: 0
```

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.

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

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

Compatible lists the unpackers that can be used in the current environment; *Incompatible* lists the unpackers that cannot be used in the current environment. An additional *Unknown* list shows the installed unpackers that might not work, for example the *vagrant* unpacker if the *vagrant* command is not found in PATH.

For example, for an experiment originally packed on Ubuntu and a user reproducing on Windows, *vagrant* is compatible (see *Vagrant Plugin*), but *installpkgs* is incompatible (we can't use Linux software packages natively).

Showing Input and Output Files

The *showfiles* command can be used to list the input and output files defined for that experiment. This is useful if you want to substitute an input file with another of your files, or get an output file out for further examination:

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

Creating a Provenance Graph

ReproZip also allows users to generate a *provenance graph* related to the experiment execution. This graph shows the relationships between files, library dependencies, and binaries during the execution. To generate such a graph, the following command should be used:

```
$ reprounzip graph package.rpz graph-file.dot
$ dot -Tpng graph-file.dot -o image.png
```

where *graph-file.dot* corresponds to the graph, outputted in the DOT language.

1.4.2 Unpacking an Experiment in Linux

There are three main unpackers specific to Linux environments: *directory*, *chroot*, and *installpkgs*. In the following, each of these unpackers are explained in detail.

Running From a 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.

Note however 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 rest of the system; in particular, should the experiment use absolute paths, they will hit the host system instead. This is fine if the system has the required packages (see *Installing Software Packages*), and the experiment's own files are addressed with relative paths.

To create the directory where the execution will take place, users should use the command *setup*:

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

where <*path*> is the diretory where the experiment will be unpacked.

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

```
$ reprounzip directory run <path>
```

which will execute the entire experiment inside the experiment directory. Users may also change the command line of the experiment by using the argument *cmdline*:

```
$ reprounzip directory 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.

Before reproducing the experiment, users also have the option to change the input files. The input files of the experiment can be listed by running the *showfiles* command (see *Showing Input and Output Files*), and then run the *upload* command:

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

where <*input-path*> is the new file's path and <*input-id*> is the input file to replace (from *showfiles*). To restore the original input file, the same command, but in the following format:

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

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, users may first list these files by running *showfiles*, and then run the *download* command:

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

where <*output-id*> is the output file to get (from *showfiles*) and <*output-path*> is the desired destination of the file. If no destination is specified, the file will be printed to stdout:

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

The experiment directory can be removed by using the *destroy* command:

```
$ reprounzip directory destroy <path>
```

Limitation: `reprounzip directory` will fail if the binaries involved in the experiment use hardcoded paths, as they will point outside the unpacked directory. The other unpackers are more reliable in that regard.

Running With *chroot*

In the *chroot* unpacker (`reprounzip chroot`), similar to *reprounzip directory*, a directory is created from the experiment package, but a full system environment is built, which can then be run with `chroot(2)` (a Linux mechanism to change the root directory `/` for the experiment to the experiment directory). Therefore, this unpacker addresses the limitation of *reprounzip directory* and does not fail in the presence of harcoded paths. It also **does not interfere with the current environment** since the experiment is isolated in that single directory.

To create the directory of the chroot environment, users should use the command *setup*:

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

where <*path*> is the diretory where the experiment will be unpacked for the chroot environment. If users run this command as root, ReproZip will restore the owner/group of the experiment files by default (unless *–no-preserve-owner* is used), and will mount your `/dev` and `/proc` directory inside the chroot (unless `--dont-mount-magic-dirs` is used).

The commands to replace input files, reproduce the experiment, and copy output files are the same as for `reprounzip directory`:

```
$ reprounzip chroot upload <path> <input-path>:<input-id>
$ reprounzip chroot run <path> --cmdline <new-command-line>
$ reprounzip chroot download <path> <output-id>:<output-path>
```

To remove the chroot environment, users can execute the command *destroy*:

```
$ reprounzip chroot destroy <path>
```

which unmounts `/dev` and `/proc` from the experiment directory and then removes the directory.

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

Installing Software Packages

By default, ReproZip identifies if the current environment already has the required software packages for the experiment, using the installed ones; 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 to try installing all the dependencies of the experiment in their environment by using the *installpkgs* unpacker (`reprounzip installpkgs`). This unpacker currently works for Debian and Debian-based operating systems only (e.g.: Ubuntu), and uses the dpkg package manager to automatically install all the required software packages directly on the current machine, thus **interfering with this 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**.

Note that 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.

1.4.3 Additional Unpackers

ReproZip has some plugins for the *reprounzip* component that provide a new range of unpackers for the system, even allowing a Linux experiment to be reproduced in different environments (e.g.: Mac OS X and Windows). These plugins do not come builtin with *reprounzip* and need to be installed separately, **after** installing *reprounzip*.

Vagrant Plugin

The *reprounzip-vagrant* plugin allows an experiment to be unpacked and reproduced using a virtual machine created through 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 is installed in the current environment.

To create the virtual machine for an experiment package, the *setup* command should be used:

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

where *<path>* is the destination directory for the Vagrant virtual machine.

The commands to replace input files, reproduce the experiment, and copy output files are the same as other unpackers:

```
$ reprounzip vagrant upload <path> <input-path>:<input-id>
$ reprounzip vagrant run <path> --cmdline <new-command-line>
$ reprounzip vagrant download <path> <output-id>:<output-path>
```

Users can also suspend the virtual machine (without destroying it) by using the *suspend* command:

```
$ reprounzip vagrant suspend <path>
```

After suspended, the virtual machine can be resumed by using the *setup/start* command.

To destroy the virtual machine, the following command must be used:

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

Docker Plugin

ReproZip can also extract and reproduce experiments using Docker containers. The *reprounzip-docker* plugin is the one responsible for such integration and it assumes that Docker is already installed in the current environment.

To create the container for an experiment package, the following command should be used:

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

where <path> is the destination directory for the Docker files.

The commands to replace input files, reproduce the experiment, and copy output files are the same as in previous unpackers:

```
$ reprounzip docker upload <path> <input-path>:<input-id>
$ reprounzip docker run <path> --cmdline <new-command-line>
$ reprounzip docker download <path> <output-id>:<output-path>
```

To destroy the container, the following command must be used:

```
$ reprounzip docker destroy <path>
```

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 Developer's Guide

Coming Soon!

CHAPTER 2

Links

- Project website
- Github repository