
ReproZip Documentation

Release 0.7

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

September 11, 2015

1	Contents	3
1.1	Why ReProZip?	3
1.2	Installation	3
1.3	Using <i>reprozip</i>	7
1.4	Using <i>reprounzip</i>	11
1.5	Frequently Asked Questions	17
1.6	Developer's Guide	21
2	Links	25

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.

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, 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 **repronzip** (for the unpack step). Additional plugins are also provided for *repronzip*: **repronzip-vagrant**, which unpacks the experiment in a Vagrant virtual machine, and **repronzip-docker**, which unpacks the experiment in a Docker container; 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 is recommended to run ReproZip. Older versions should allow *reprounzip* to work, but some features will not be available ¹. If you don't have Python on your machine, you can get it from python.org ²; you should prefer a 2.x release to a 3.x one. You will also need the [pip](https://pip.pypa.io/en/stable/) 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
<i>reprozip</i>	SQLite ³ , Python headers ⁴ , a working C compiler
<i>reprounzip</i>	None
<i>reprounzip-vagrant</i>	Python headers ⁴ ⁵ , a working C compiler ⁵ , Vagrant 1.1+ , VirtualBox
<i>reprounzip-docker</i>	Docker

Installing *reprozip*

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

```
$ pip install reprozip
```

To update the software, use the flag `-U`:

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

Installing *reprounzip*

To install the *reprounzip* component, simply run the following command:

```
$ pip install reprounzip
```

To update the software, use the flag `-U`:

```
$ pip install -U reprounzip
```

The additional plugins for *reprounzip* can also be installed using the same command:

```
$ pip install reprounzip-docker reprounzip-vagrant
```

Alternatively, you can install *reprounzip* with all the available plugins using:

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

¹ *reprounzip* graph will not work due to Python bug 13676 related to `sqlite3`.

² On Debian and Debian-based, you can use `sudo apt-get install python`.

³ On Debian and Debian-based, you can use `sudo apt-get install libsqlite3-dev`.

⁴ On Debian and Debian-based, you can use `sudo apt-get install python-dev`.

⁵ Required to build `PyCrypto`.

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 is recommended to run ReproZip. Older versions should allow *reprounzip* to work, but some features will not be available ⁶. 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
<i>reprounzip</i>	None
<i>reprounzip-vagrant</i>	Python headers ^{7 8} , a working C compiler ^{7 8} , Vagrant 1.1+, VirtualBox
<i>reprounzip-docker</i>	Docker

See also:

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

Installing *reprounzip*

First, be sure to upgrade *setuptools*:

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

To install the *reprounzip* component, simply run the following command:

```
$ pip install reprounzip
```

To update the software, use the flag `-U`:

```
$ pip install -U reprounzip
```

The additional plugins for *reprounzip* can also be installed using the same command:

```
$ pip install reprounzip-docker reprounzip-vagrant
```

Alternatively, you can install *reprounzip* with all the available plugins using:

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

⁶ *reprounzip* graph will not work due to Python bug 13676 related to sqlite3.

⁷This is usually provided by installing Xcode (in the Mac App Store) and the Command Line Developer Tools; instructions on installing the latter may depend on your Mac OS X version (some information on StackOverflow [here](#)).

⁸Required to build PyCrypto.

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 is recommended to run ReproZip. Older versions should allow *reprounzip* to work, but some features will not be available⁹. 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
<i>reprounzip</i>	None
<i>reprounzip-vagrant</i>	PyCrypto ¹⁰ , Vagrant 1.1+, VirtualBox
<i>reprounzip-docker</i>	Docker

See also:

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

Installing *reprounzip*

To install the *reprounzip* component, simply run the following command:

```
$ pip install reprounzip
```

To update the software, use the flag `-U`:

```
$ pip install -U reprounzip
```

The additional plugins for *reprounzip* can also be installed using the same command:

```
$ pip install reprounzip-vagrant
$ pip install reprounzip-docker
```

Alternatively, you can install *reprounzip* with all the available plugins using:

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

1.2.4 Anaconda

reprozip and *reprounzip* can also be installed on the [Anaconda](#) Python distribution, from Binstar:

⁹ *reprounzip* graph will not work due to [Python bug 13676](#) related to sqlite3.

¹⁰ A working C compiler is required to build PyCrypto. For installation without building from source, please see [this page](#).

```
$ conda install -c https://conda.binstar.org/vida-nyu reprozip reprozip reprozip-docker reprozip
```

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.

Multiple runs can also be traced and combined in a single package by using the flag `--continue`, or `-c`, for all the runs except the first one:

```
$ 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 Debian or Debian-based (e.g.: Ubuntu), *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 do not 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, 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, 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>
- 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 that you can prevent *reprozip* from guessing which files are inputs or outputs using the `--dont-find-inputs-outputs` flag.

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 `reprozip graph`; please refer to *Creating a Provenance Graph* for more information.

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. Alternatively, users can trace each run individually and include them all in the same package, but since one may reproduce individual runs, this does not ensure that the server will be alive when reproducing the experiment.

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:

```
$ repronzip -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 *repronzip* 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 *repronzip-vagrant* plugin) is compatible, but *installpks* 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 `repronzip 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:

```
$ repronzip 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:

```
$ repronzip -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)
```

This 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.

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 `reprounzip graph` command should be used:

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

where *graph-file.dot* corresponds to the graph, outputted in the DOT language. You can use [Graphviz](#) to load and visualize the 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`.

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 that 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 ununpacker is described in more details; for more information on how to use an ununpacker, please refer to *Using an Unpacker*.

The *directory* Unpacker: Unpacking as a Plain Directory

The *directory* ununpacker (`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 ununpacker 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 ununpacker 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 ununpacker 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 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 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 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.

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>
```

Note: This unpacker is **not** distributed with *reprounzip*; it is a separate package that should be installed before using (see [reprounzip-vagrant plugin](#)).

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.

Note: This unpacker is **not** distributed with *reprounzip*; it is a separate package that should be installed before using (see [reprounzip-docker plugin](#)).

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.

Setting Up an Experiment Directory

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.

Note: Most unpackers assume an Internet connection for the `setup` command and will be downloading required software from the Internet. Make sure that you have an Internet connection, and that there is no firewall blocking the access.

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), you need to provide the id of the run to be reproduced:

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

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.

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:

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

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

```
$ repronzip showfiles <path>
Input files:
  program_config
    (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:

```
$ repronzip 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 no destination is specified, the file will be printed to stdout:

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

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).

See also:

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

1.4.4 Further Considerations

Reproducing Multiple Execution Paths

The `reprozip` component can only guarantee that `repronzip` 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 Frequently Asked Questions

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

ReproZip uses some heuristics to identify an input or output file. However, this is only intended to be a starting point: 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.

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

This is probably the case where these output files do 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 `output_files` 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.

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

1.5.3 Why aren't any files packed when tracing a daemon?

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.

However, 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.

1.5.4 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 in your experiment 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 *Further Considerations When Packing* for more information.

1.5.5 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/**
```

Also 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).

1.5.6 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.

1.5.7 Can ReproZip pack graphical tools?

Yes! On Linux, graphical display is handled by the X server, to which applications can connect 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. Note that 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.

1.5.8 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.

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 accessing 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.

1.5.9 What if my experiment runs on a distributed environment?

ReproZip cannot trace across multiple machines. You could trace each component separately, but ReproZip has no support yet to setup these multiple machines in the right way from the multiple `.rpz` files. In particular, you will probably need to set up the same network for the components to talk to each other.

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

Python is trying to build `PyCrypto`, one of the dependencies of *reprounzip-vagrant*, but there is no C compiler available. You can either build `PyCrypto` from source, or follow the instructions on [this website](#) to get the non-official binaries.

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

This is an issue with the Apple LLVM compiler, which treats unrecognized command-line options as errors. As a workaround, before installing *reprounzip-vagrant*, run the following:

```
$ sudo -s
$ 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]
```

1.5.12 Why are there warnings from `requests/urllib3`?

You may be seeing warnings like this:

```
/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.org/en/latest/security.html#insecureplatformwarning.
```

Most Python versions are insecure, because they do not validate SSL certificates. Python 2.7.9 and later shouldn't be affected, but if you see `InsecurePlatformWarning`, you can run `pip install requests[security]`, which should bring in the missing components.

1.5.13 Why does the experiment fail with `Error: Can't open display: :0?`

The experiment probably involves running a GUI tool. 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.

1.5.14 Why does *reprounzip directory* fail with `IOError`?

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. 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.

1.5.15 Why does *reprounzip* fail with `DistributionNotFound` errors?

You probably have some plugins left over from a previous installation. Be sure to upgrade or remove outdated plugins when you upgrade *reprounzip*.

The following command might help:

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

1.5.16 Why does *reprounzip* show running in `chroot`, ignoring request?

This message comes from the `systemd` client, which will probably not work with *ReproZip*. In this case, the experiment should be re-packed without using `systemd` (see [this question](#) for more information).

1.5.17 Why does *reprounzip vagrant setup* fail to resolve a host address?

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. Please make sure that your anti-virus/firewall is not causing this issue.

1.5.18 Why does *reprounzip run* fail with `no such file or directory` or similar?

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

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. You may want to try using the packed files directly to ensure compatibility. Also, note that, while packing, the user can choose not to include some packages, meaning that *reprounzip* will have to install the one from the package manager, which, again, is not guaranteed to be compatible. In this case, try contacting the author of the *ReproZip* package.

When using `reprounzip vagrant` and `reprounzip docker`, *ReproZip* tries to detect the closest base system for unpacking the experiment. You may also want to 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.

1.6 Developer's Guide

1.6.1 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 due to a lot of operations that are difficult to cover on *Travis* (*Vagrant* VMs and *Docker* containers 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 reprozip-dev@vgc.poly.edu.

1.6.2 Writing Unpackers

ReproZip is divided into two steps. Packing gives a generic package containing the trace SQLite database, the YAML configuration file (listing 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 turned into a runnable form by *repronzip*. 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.

The ViDA group maintains different unpackers: the defaults ones (`directory` and `chroot`), `vagrant` (distributed as `repronzip-vagrant`) and `docker` (distributed as `repronzip-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 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* `repronzip.unpackers`. The name of the *entry_point* (before `=`) will be the *repronzip* subcommand, and the value is a callable that will get called with the `argparse.ArgumentParser` object for that subcommand.

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

Example of `setup.py`:

```
setup(name='repronzip-vagrant',
      namespace_packages=['repronzip', 'repronzip.unpackers'],
      install_requires=['repronzip>=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 are free to 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`.

1.6.3 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: *reprozip-dev@vgc.poly.edu*.

Links

- Project website
- GitHub repository
- Mailing list: reprozip-users@vgc.poly.edu