ReproZip Documentation

Release 0.5

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

November 24, 2014

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 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.

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. 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. 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, the *reprozip* component 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 ReproZip components is as follows:

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

Python 2.7.3 or greater [3] is required to run ReproZip. If you don't have it yet on your machine, you can get it from python.org; you should prefer a 2.x releases to 3.x [4]. 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 [2], Python headers [2], a working C compiler
reprounzip	None
reprounzip-vagrant	pycrypto [6], Vagrant, VirtualBox
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, as mentioned before. Then, to install a ReproZip component, simply run the following command:

```
$ pip install reprozip
$ # or:
$ pip install 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]
```

1.3 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

[1]By installing additional unpackers; the default bundled unpackers only work on Linux.

[2]By installing additional unpackers; the default bundled unpackers only work on Linux.

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

[4] On Debian and Debian-based, these are provided by *python*, *python-dev*, and *libsqlite3-dev*.

[5]Building PyCrypto on POSIX will require a C compiler and the Python development headers (*python-dev* package on Debian and derived). For installation on Windows without building from source, please see here.

[6]Building PyCrypto on POSIX will require a C compiler and the Python development headers (*python-dev* package on Debian and derived). For installation on Windows without building from source, please see here.

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*. 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 an experiment:

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

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 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 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 trace` 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 generated by the tracer and drives the packing step.

You possibly do not need to edit it, as the automatically-generated file should be 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, 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 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 `argv`, and add or remove environment variables by editing `environ`. Besides, `input_files` and `output_files` can be modified to inform ReproZip about any input/output file that the tool may have failed in detecting, and also to give meaningful id names to these files (this may be useful for the unpacking step). Other attributes should mostly not be changed

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 `reprozip trace` command, 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, 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 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* to `reprozip trace`.

Packing GUI and Interactive Tools

Currently, ReproZip cannot ensure that GUI interfaces will be correctly reproduced, 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

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. 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 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 default name) that always points to the latest result, and use that as the output file's path in the configuration file (under the `output_files` 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). You can use `-v` for *verbose* 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:
    ...
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. An additional *Unknown* list shows the installed unpackers that might 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 the *vagrant* command is not found 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 choosen 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.

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 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 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 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.

Limitation: `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*.

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 harcoded 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 pseudofiles (these can be restored by rebooting or running the `MAKEDEV` script).

Limitation: 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**.

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.

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 entire 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.

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 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:

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

```
$ reprounzip 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).

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 Frequently Asked Questions

1.5.1 Why *reprozip* does not identify my input/output file?

ReproZip uses some heuristics to determine what is and what is not an input or output file. However, this is intended to be a starting point: you should check the configuration file (`input_files` and `output_files` sections) and add/remove paths there; giving readable id names to input/output files, such as *database-log* or *lookup-table* also helps.

1.5.2 Why *reprounzip* cannot 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 no files get 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.

Note, however, that running GUI tools (connecting to an X server) is yet not supported by ReproZip.

1.5.7 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.

1.5.8 What if I need to pack multiple command lines?

The easiest way, in this case, is to write a script that runs all the desired command lines, and then to trace the execution of this script with *reprozip*.

1.5.9 Why I am having issues with *reprounzip-vagrant* on Python 3?

The *reprounzip-vagrant* plugin is compatible with Python 3; however, the **scp.py** library used to transfer files to and from the virtual machine has a number of issues. Until the maintainer accepts our patch, you can install our fixed version from GitHub using:

```
pip install 'git+https://github.com/remram44/scp.py.git#egg=scp'
```

```
.. _distribnotfound:
```

1.5.10 Why *reprounzip* shows 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]
```

1.5.11 Why *reprounzip* shows `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).

1.6 Developer's Guide

1.6.1 General Development Information

Development happens on Github; bug reports of feature requests are welcome. If you are interested in giving 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. Note that test coverage is not very high, this is because a lot of operations are difficult to cover on Travis (Vagrant VMs and Docker containers can't be used over there).

If you have questions or need help with the development of an unpacker or plugin, don't hesitate to use the development mailing-list at *reprozip-dev@vgc.poly.edu*.

1.6.2 Writing Unpackers

ReproZip is divided in two steps. Packing gives out a generic package containing the trace SQLite database, YAML configuration file (listing the paths, packages, and run metadata like command-line, environment variables, and input/output files) and actual files, and then in a second step that pack can be turned into a runnable form by reprounzip. This allows for the selection of the unpacker to be left to the reproducer, and also means that when a new unpacker comes out, people will be able to use it on their old packs without change.

The ViDA group maintains different unpackers: the two defaults ones (`directory` and `chroot`), `vagrant` (distributed as reprounzip-vagrant) and `docker` (distributed as reprounzip-docker). However, the interface has been thought so that new unpackers could be added easily. While taking a look at the "official" unpackers' source is probably a good idea, this page gives some useful information about how they work.

Structure

An unpacker is a Python module. It can be distributed separately or be part of a bigger distribution; it just has to be 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 (the part 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 in there should you want to. Please remember to put the correct code in the `__init__.py` file (which you can copy from here) so 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 `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; these are:

- `setup`, to create the unpacked directory and set everything up for execution;
- `run`, to execute the experiment;
- `destroy`, to bring down all that setup had to prepare and delete the unpacked directory safely;
- `upload` and `download`, to either substitute input files in the experiment with your own, or get the output files out for further examination.

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, reprounzip-vagrant exposes `setup/start` which starts or resume 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 any experiment out there. Even trickier, reprounzip-vagrant needs to manipulate POSIX filenames on Windows, for example in the unpacker.

Because of that, 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 Python functions expect, keeping in mind 2/3 differences (most certainly either `filename.path` or `str(filename)`).

Unpacker directory format

Unpackers usually create a directory with all that is needed 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
```

"Official" unpackers unpack the config.yml file to the root of that directory, and keep status information in a .reprounzip file, a dict in `pickle` format. Following this same structure will allow the `showfiles` command and `FileUploader` and `FileDownloader` classes to work automatically, so you should try to stick to it.

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 in signal.py.

1.6.3 Final notes

After reading this manual, 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, don't hesitate to ask on the development mailing-list: *reprozip-dev@vgc.poly.edu*.

CHAPTER 2

Links

- Project website
- Github repository