ReproZip: Computational Reproducibility With Ease

Fernando Chirigati
New York University
fchirigati@nyu.edu

Rémi Rampin
New York University
remi.rampin@nyu.edu

Dennis Shasha
New York University
shasha@cs.nyu.edu

Juliana Freire
New York University
juliana.freire@nyu.edu

ABSTRACT

We present ReproZip, the recommended packaging tool for the SIGMOD Reproducibility Review. ReproZip was designed to simplify the process of making an existing computational experiment reproducible across platforms, even when the experiment was put together without reproducibility in mind. The tool creates a self-contained package for an experiment by automatically tracking and identifying all its required dependencies. The researcher can share the package with others, who can then use ReproZip to unpack the experiment, reproduce the findings on their favorite operating system, as well as modify the original experiment for reuse in new research, all with little effort. The demo will consist of examples of non-trivial experiments, showing how these can be packed in a Linux machine and reproduced on different machines and operating systems. Demo visitors will also be able to pack and reproduce their own experiments.

1. INTRODUCTION

Reproducibility of computational experiments across platforms and time brings a range of benefits to science. First, reproducibility enables reviewers to test the outcomes presented in papers. Second, it allows new methods to be objectively compared against methods presented in reproducible publications. Third, researchers are able to build on top of previous work directly. Last but not least, recent studies indicate that reproducibility increases impact, visibility, and research quality [1, 26], and helps defeat self-deception [19].

In spite of its importance, achieving reproducibility has proved elusive for computational experiments, and this has led to a credibility crisis [10]. Ideally, researchers should create a compendium that encompasses all the components that are required to reproduce the experiment's results, including data, code, software, library dependencies, and information about the source computational environment (e.g., operating system and hardware architecture), i.e., the *provenance of the experiment*. As a computational experiment may consist of intricate chains of dependencies, manually creating such a compendium is a hard task at best and barely feasible most of the time.

There has been work on a plethora of tools that help create reproducible experiments and capture their provenance. Scientific workflow systems [6] represent an experiment as a dataflow (or workflow), stitching together its different steps and connecting data and processes in an executable representation. But adapting an experiment to run in these systems requires a steep learning curve that has hampered their adoption. Configuration management tools [3, 21, 22] automate the configuration of an experiment (e.g., installation of dependencies) by supporting the creation of "recipes" that can be reused every time a new machine needs to be configured; these scripts, however, need to be generated manually by researchers. Virtual machines (VMs) can be used to encapsulate the entire context of an experiment, providing an exact replica of the computational environment where the experiment took place. A drawback of virtual machines is that the derived images can be large, including a number of files that are not related to the experiment. Lighter-weight solutions exist for deploying and maintaining software containers, such as Docker [9], but similar to VMs, the researcher is faced with the burden of ensuring that all necessary dependencies are in the container. Computational platforms, such as Burrito [14] and Arnold [8], capture the provenance of all processes at the operating system (OS) level, recording detailed state information, but only allow experiments to be replayed in the same system. Last, there is a class of tools aimed at particular domains, e.g., GenePattern [12] is a genomic analysis platform, Madagascar [16] is used to analyze seismic data, Sumatra [7] is used for numerical computations, and noWorkflow [18] supports Python scripts only.

When using any of these solutions, making *existing* experiments reproducible is a time-consuming and error-prone process: researchers need to manually deploy dataflows, create configuration recipes, or transfer all the provenance to a VM. This difficulty discourages researchers from creating reproducible experiments, especially if the reason is solely for publication purposes [2, 5, 24].

In contrast to these approaches, `ReproZip` [4] provides a *lightweight* solution that makes experiments reproducible *without forethought*. Researchers can create an experiment without thinking about reproducibility and use `ReproZip` to make it reproducible and portable to other machines immediately or many years later. `ReproZip` has been recommended for the SIGMOD Reproducibility Review.[1]

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

[1] http://db-reproducibility.seas.harvard.edu/

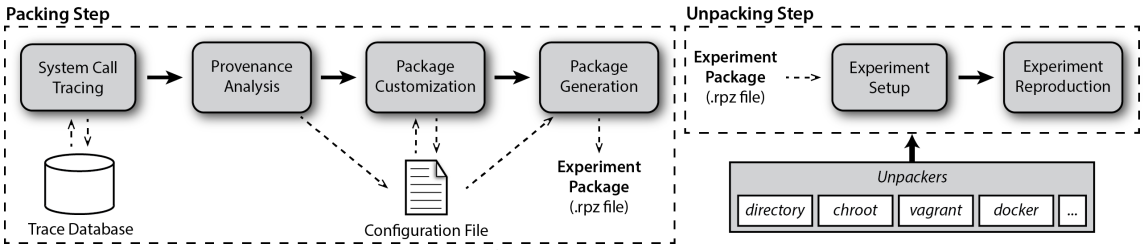

Figure 1: Architecture of `ReproZip`.

There are packaging systems that can help with reproducibility in the same *without-forethought* spirit as `ReproZip`, including CDE [13], PTU [20], and CARE [15]. They have the ability to create self-contained packages for the reproduction of existing Linux-based experiments. They do so by tracing the execution, intercepting system calls, and getting information related to the experiment. Using this provenance, all the necessary components of the experiment are copied to a package, which can then be sent to someone who wants to reproduce it. While reproducing an experiment, these tools dynamically change the system calls to point to the correct files included in the package.

Similar to these tools, `ReproZip` *automatically* and *transparently* captures the provenance of an existing experiment by tracing system calls, and uses this information to create a lightweight reproducible package that includes only the required files needed for its reproduction. However, `ReproZip` adds important features and contributions:

- *Portability:* Existing packaging systems have limited portability, as experiments can only be reproduced in environments with a compatible Linux kernel. In contrast, `ReproZip` provides *unpackers* that allow researchers to automatically create a VM or a Docker container encompassing the experiment, thus allowing it to be reproduced in different OS'es. `ReproZip` also generates a workflow specification for the experiment, which can be used to easily change parameters or modify the original dataflow.
- *Extensibility:* By implementing new unpackers, researchers can easily extend `ReproZip` to port experiments to other environments and systems (e.g., configuration "recipes" could be generated) while keeping compatibility with existing packaged experiments.
- *Modifiability:* `ReproZip` automatically identifies input files, parameters, and output files, allowing researchers to easily *modify* these for reuse purposes.
- *Usability:* Researchers have control over the collected trace and can customize the reproducible package. `ReproZip` also provides command-line interfaces that make it easier to setup, reproduce, and modify the original experiment.

The first (beta) version of `ReproZip` was developed in 2013 [4]. Since then, we have redesigned the system to be more general and cover a wider range of reproducibility requirements, and also to substantially improve its usability. In this demo, we describe `ReproZip` version 1.x. `ReproZip` can be downloaded from `https://vida-nyu.github.io/reprozip`, and a tutorial video on how to use the system is available at `https://bit.ly/1N9QHCs`.

2. SYSTEM ARCHITECTURE

`ReproZip` supports two main tasks (Figure 1): *packing*, which captures the dependencies for an experiment and creates a self-contained package, and *unpacking*, which supports the extraction of the package's content and the reproduction of the original experiment. In the following, assume a computational experiment E that runs on a single machine on OS S and that can be executed by running command line c.

2.1 Packing Experiments

The packing step is accomplished by four main modules: System Call Tracing, Provenance Analysis, Package Customization, and Package Generation (Figure 1). `reprozip` is the command-line tool responsible for this step.

System Call Tracing. To create a reproducible package, `ReproZip` needs to identify all the dependencies for E. Command c is prepended with `reprozip trace`, which runs E while tracing all the system calls; since this is done through `ptrace`, S must (currently) be a Linux-based OS. By tracing system calls, `ReproZip` can transparently capture all the provenance of E. For instance, `execve` is used to execute programs, `open` informs `ReproZip` which files are being opened, and `read` and `write` read from and write to a file, respectively. Therefore, `ReproZip` collects a plethora of information regarding E, including command-line arguments, environment variables, files read, and files written, and stores everything in a SQLite database.

The previous version of `ReproZip` used SystemTap [23] for tracing and MongoDB [17] for storage. While SystemTap is complex to install and update, `ptrace` is part of the Linux kernel. SQLite is also much lighter than MongoDB and part of the Python distribution. These changes were made to enhance usability, as the packing step no longer has external requirements.

Provenance Analysis. The provenance data is analyzed to detect the required components of E. For instance, given the files that were read and using the package manager of the OS, `ReproZip` can identify the *software packages* on which E depends. `ReproZip` also uses some heuristics to identify *input and output files*: files that were only read but that do not belong to any software package are considered input files; files that were only written are considered output files; files that were read and then written are considered stateful files (e.g., log or database files). This significantly improves the beta version of `ReproZip`, in which the provenance analysis is limited to detecting explicit input/output files and parameters from c. All the collected information is then written to a human-readable *configuration file*, which can be edited by researchers.

Package Customization. The configuration file contains all the required information for packing E, including: execution information (e.g., command, arguments, working directory, environment variables, and OS information), input files, output files, and identified software packages. Even though this automatically-generated file is sufficient to create a working *experiment package*, researchers may edit this file (i) to modify command-line arguments; (ii) to modify

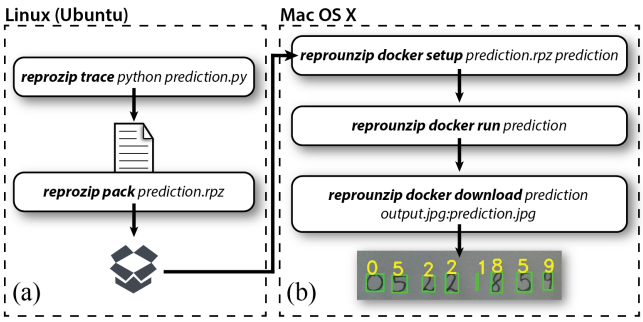

Figure 2: Packing (a) and unpacking (a) a computer vision experiment with `ReproZip`.

environment variables; (iii) to remove big files that can be obtained elsewhere; (iv) to remove sensitive or proprietary information; or (v) to exclude software packages.

Package Generation. After tracing all the executions from E, analyzing the provenance data, and optionally editing the configuration file, the *experiment package* P can be created by running `reprozip pack`. The package is a `.rpz` file, which encompasses all the required data and information for reproducing the traced execution (e.g., input files, library dependencies, and binaries). P can then be sent to others so that E can be reproduced.

Example. *Figure 2(a) shows an experiment on Ubuntu that predicts the values of hand-writing digits from an input image and outputs a new image with the predictions. Packing this experiment is as easy as running two commands. While the full VM takes 4GB,* `prediction.rpz` *has 67MB, which is substantially lighter-weight for sharing.*

2.2 Unpacking Experiments

Given P, E can be reproduced by using the command-line tool `reprounzip`. Different *unpackers* are provided to reproduce the corresponding experiment.

2.2.1 Unpackers

The choice of an unpacker depends on the OS S' where E will be reproduced: S' can be either compatible with S ($S' \approx S$), or completely different from S ($S' \not\approx S$). An example of the former is when S' and S have compatible Linux kernels, while the latter scenario happens when S' is a Windows or a Mac OS X machine. Portability is a significant new feature of `ReproZip`, which previously could only unpack and reproduce experiments across Linux systems; more specifically, we only supported what is now the *directory* unpacker (see below) in our beta version [4].

$S' \not\approx S$. There are two unpacking options: the *vagrant* and the *docker* unpackers. The *vagrant* unpacker allows E to be unpacked and reproduced inside a virtual machine automatically created through Vagrant [25]. Therefore, E can be reproduced on any system S' supported by this tool. The *docker* unpacker, on the other hand, unpacks and reproduces E in a Docker [9] container, either on the researcher's own machine, a remote one, or on a cloud provider. In both cases, `ReproZip` automatically selects a base system image that is the closest to S, but researchers can also explicitly choose their own.

$S' \approx S$. `ReproZip` supports two additional unpackers: *directory* and *chroot*. The former unpacks E in a single directory: E can be reproduced 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 original system S. Note that the created directory is *not* isolated from S'. In particular, should E use hardcoded absolute paths, they will hit the host system instead. This unpacker is meant for relatively well-behaved experiments that address files with relative paths only.

In the *chroot* unpacker, similar to the *directory* one, a directory is created from P. However, a full system environment is also built, which is then run with `chroot`, a Linux mechanism that changes the root directory for E to the created directory. Therefore, this unpacker works even in the presence of hardcoded absolute paths. Note as well that the unpacked software does not interfere with S' since E is isolated in that single directory.

`ReproZip` also has an option to natively install all the software packages. In this case, when reproducing E, both *directory* and *chroot* unpackers can use the installed dependencies, instead of the ones copied from inside P.

Extensions. If researchers need to port E to a different system not currently supported by `ReproZip`, they can easily create their own unpacker and attach it to the tool. `ReproZip` was redesigned in a way that makes it easy to create new extensions that will still support every package created in the past.

2.2.2 Modules

The unpacking step, regardless of the unpacker, consists of two main modules: Experiment Setup and Experiment Reproduction (Figure 1).

Experiment Setup. First, given P, the experiment needs to be extracted, and this is accomplished by running the `setup` command. While, for the *directory* and *chroot* unpackers, this command means copying E to a single directory (and creating a full system environment for the latter), a virtual machine and a Docker container are initialized with E for the *vagrant* and *docker* unpackers, respectively.

Experiment Reproduction. Once the package has been extracted, E can be reproduced by using the `run` command. The way E is reproduced depends on the chosen unpacker: for *directory*, the execution happens inside the experiment directory; for *chroot*, it happens inside the created file system; and for *vagrant* and *docker*, it is done inside the virtual image and the container, respectively. All the interaction with E is done through `reprounzip` and its command-line interfaces. Researchers do not need to know how to reproduce E or even how to use Vagrant or Docker, since `reprounzip` will perform the necessary steps automatically for them according to the selected unpacker. Such command-line interfaces were nonexistent in `ReproZip`'s beta version.

Example. *Figure 2(b) shows how to unpack—using the docker unpacker—*`prediction.rpz` *on a Mac OS X machine. The* `setup` *and* `run` *commands are the only steps necessary for reproducing the original experiment. Additionally, the command* `download` *(Section 2.2.3) may be used for retrieving the output file. The automatically-generated Docker image has 500MB. When using the vagrant unpacker, the virtual machine takes up 1.40GB, much lighter when compared to the original one (4GB) since the former encompasses only the files necessary for the reproduction.*

2.2.3 File and Dataflow Management

Recall that input and output files are identified based on heuristics. In the unpacking step, `ReproZip` uses this information to let researchers alter input files and retrieve the output files after E is reproduced. The command `showfiles` can be used to see which files are input and output, and the commands `upload` and `download` can be used to replace an input file and to retrieve an output file, respectively. This makes it easy to *reuse* E, or evaluate it on different inputs.

`ReproZip` can also derive a specification of the experiment workflow. The main programs of the experiment are wrapped in workflow modules that automatically take the command-line arguments and input files as inputs. Currently, `ReproZip` derives workflows that can be run on VisTrails [11]. Using this system, (i) the dataflow of an experiment can be understood; (ii) unpacked experiments can be executed; (iii) researchers can explore experiments and try different parameters and input files; and (iv) researchers can extend the original workflow to explore different techniques and perform analyses, or modify the dataflow to reuse steps for their own research.

3. DEMONSTRATION

In our demonstration, we encourage visitors to bring their own experiments in their Linux machines. They will:

- Install `ReproZip` on their machine S;
- Trace the execution of their experiment;
- Create a reproducible package;
- Transfer the package to a completely different machine S' (either a Mac OS X or a Windows machine);
- Reproduce the experiment on S' and modify its inputs;
- Generate a VisTrails workflow to visualize the experiment dataflow and alter its pipeline.

In addition to the visitors' own experiments, we will have a number of real experiments from different domains available, including computer vision, visualization (e.g., interactive and exploratory tools), and database research.

4. DISCUSSION AND FUTURE WORK

For experiments that run on Linux systems, `ReproZip` constitutes a simple-to-deploy tool that makes an experiment reproducible across different platforms and long after it was created. Therefore, "*It's too difficult to reproduce*" can no longer be used as an excuse. We believe this greatly contributes to scientific quality, allows researchers to reuse the experiment and software produced by others, helps analyze the pipeline and dependencies of an experiment, and even helps industrial software developers deploy their software on new OS'es.

Currently, `ReproZip` supports a wide range of experiments, including client-server scenarios, experiments with databases, and graphical and interactive tools. Also, if experiment E is composed by multiple command-line executions (e.g., a pipeline, or different execution paths), each execution can be traced separately and the analyzed information can be added to a single configuration file, so that all the executions are included in the same package P. When unpacking P, researchers can select which runs to reproduce.

For future development, we are working on supporting experiments that run on distributed environment (e.g., MPI and Hadoop clusters). Also, we plan on supporting non-Linux systems for the packing step.

5. REFERENCES

[1] C. G. Begley and L. M. Ellis. Drug development: Raise standards for preclinical cancer research. *Nature*, 483(7391):531–533, 2012.

[2] P. Bonnet et al. Repeatability and Workability Evaluation of SIGMOD 2011. *SIGMOD Rec.*, 40(2):45–48, 2011.

[3] Chef. `https://www.chef.io/solutions/configuration-management/`.

[4] F. Chirigati, D. Shasha, and J. Freire. Packing Experiments for Sharing and Publication. In *SIGMOD '13*, pages 977–980, 2013.

[5] C. Collberg, T. Proebsting, and A. M. Warren. Repeatability and Benefaction in Computer Systems Research. Technical Report TR 14-04, University of Arizona, 2015.

[6] S. B. Davidson and J. Freire. Provenance and Scientific Workflows: Challenges and Opportunities. In *SIGMOD '08*, pages 1345–1350, 2008.

[7] A. Davison. Automated Capture of Experiment Context for Easier Reproducibility in Computational Research. *Computing in Science Engineering*, 14(4):48 –56, july-aug. 2012.

[8] D. Devecsery, M. Chow, X. Dou, J. Flinn, and P. M. Chen. Eidetic systems. In *OSDI'14*, pages 525–540, 2014.

[9] Docker. `https://www.docker.com/`.

[10] D. Donoho, A. Maleki, I. Rahman, M. Shahram, and V. Stodden. Reproducible research in computational harmonic analysis. *Computing in Science & Engineering*, 11(1):8–18, Jan.-Feb. 2009.

[11] J. Freire, D. Koop, E. Santos, C. Scheidegger, C. T. Silva, and H. T. Vo. *The Architecture of Open Source Applications*, chapter VisTrails. Lulu Publishing, Inc., 2011.

[12] GenePattern. `http://www.broadinstitute.org/cancer/software/genepattern/`.

[13] P. Guo. CDE: A Tool For Creating Portable Experimental Software Packages. *Computing in Science & Engineering*, 14:32–35, 2012.

[14] P. J. Guo and M. Seltzer. BURRITO: Wrapping Your Lab Notebook in Computational Infrastructure. In *TaPP'12*, pages 7–7, 2012.

[15] Y. Janin, C. Vincent, and R. Duraffort. CARE, the Comprehensive Archiver for Reproducible Execution. In *TRUST '14*, pages 1:1–1:7, 2014.

[16] Madagascar. `http://www.ahay.org/wiki/Main_Page`.

[17] MongoDB. `http://www.mongodb.org/`.

[18] L. Murta, V. Braganholo, F. Chirigati, D. Koop, and J. Freire. noWorkflow: Capturing and Analyzing Provenance of Scripts. In *IPAW*, pages 71–83, 2015.

[19] R. Nuzzo. How scientists fool themselves, and how they can stop. *Nature*, 526(7572):182–185, 2015.

[20] Q. Pham, T. Malik, and I. Foster. Using Provenance for Repeatability. In *TaPP '13*, pages 2:1–2:4, 2013.

[21] Puppet. `http://puppetlabs.com/`.

[22] C. Ruiz, O. Richard, and J. Emeras. Reproducible Software Appliances for Experimentation. In *Testbeds and Research Infrastructure: Development of Networks and Communities*, volume 137, pages 33–42. 2014.

[23] SystemTap. `http://sourceware.org/systemtap/`.

[24] C. Tenopir, S. Allard, K. Douglass, A. U. Aydinoglu, L. Wu, E. Read, M. Manoff, and M. Frame. Data Sharing by Scientists: Practices and Perceptions. *PLoS ONE*, 6(6), 2011.

[25] Vagrant. `https://www.vagrantup.com/`.

[26] P. Vandewalle, J. Kovacevic, and M. Vetterli. Reproducible Research in Signal Processing. *Signal Processing Magazine, IEEE*, 26(3):37–47, 2009.