Reproducible Experiments for Internet Systems

Frank Cangialosi
Akshay Narayan
MIT CSAIL
USA

Abstract

We relate our experience and recommendations from the authors' perspective in the artifact evaluation (AE) process, which is becoming increasingly common in systems venues.

CCS Concepts

• **Networks** → *Middle boxes / network appliances*; • **General and reference** → **Evaluation**.

Keywords

Reproducibility, Internet Systems

ACM Reference Format:
Frank Cangialosi and Akshay Narayan. 2022. Reproducible Experiments for Internet Systems. In *Proceedings of the 5th International Workshop on Practical Reproducible Evaluation of Computer Systems (P-RECS '22), June 30, 2022, Minneapolis, MN, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3526062.3536355

1 Introduction

In recent years, the concept of an (optional) artifact evaluation (AE) process has gained traction in the systems community (e.g., SOSP, OSDI, and EuroSys [1]). Authors of accepted conference papers may submit an artifact associated with their paper and a team of reviewers then evaluates the availability and reproducibility of the artifact. The results of this process are stamped onto the official conference PDF.

However, while most would agree that the ability to reproduce the results in a research paper is nice in theory, the reality is that many authors do not go through this process in practice. In 2021, 177 papers were published at EuroSys, OSDI, and SOSP; of these, only 68% submitted an artifact, and only 48% (of those submissions) received the "Results Reproducible" badge (the highest of three awarded) from the artifact evaluation process. One reason for this may be that the artifact evaluation deadline is often on the order of a few days (e.g. for EuroSys 2021, it was 3 days) after the notification of paper acceptance. Packaging an artifact within this short time window is a challenge. Thus, waiting until paper acceptance to start packaging is probably too late. If the starting point is a haphazard collection of scripts (or, as we all have performed from time to time, a search through shell history), submitting a good artifact will be challenging.

Packaging artifacts is hard because a project's code and experiments are often not easily portable—in systems projects, the experiment environments are complex, there are fewer abstractions one can rely on, and everything from details in the kernel implementation to the randomness of running experiments on the Internet can ultimately impact results. Incorporating reproducibility in the project from the beginning can amortize this cost. More importantly, *reproducibility can improve the quality of a research project and its usefulness to the broader community*:

- During the research process, we generate insights and evolve our systems based on the results of experiments. Unintentionally running the system slightly differently each time could yield inconsistent results and incorrect insights.
- One way to measure a project's reproducibility is to consider how many future support emails others trying to use your system send, and more importantly how long it takes to help them. For example, once, a colleague who wanted to evaluate some prior work in a new setting spent several weeks going back and forth with authors just to properly tune the parameters and recreate the graphs.
- A key to impactful work is whether or not it helps spur further research. If no one can run your artifact, it will be difficult to build on or even compare against it as a baseline. Further, future users may not be willing to spend as much time as the artifact evaluation committee, who have dedicated time to try their best to get your artifact to work. This may be months or years after the initial system development, in which case the environment and dependencies may have changed significantly. If this is not well documented, it may be impossible to reproduce the configuration.
- Experiments can serve as documentation of interesting ways to use the system; future users can branch off from the artifact's experiments for their use cases.

In this paper, we discuss[1] how to incorporate reproducibility in the process of systems and networking research. In particular, we recount our experience thinking about reproducibility for our EuroSys '21 paper, Bundler [5], and share some of the components we used that we hope will be relevant for other projects.

2 Principles and Getting Started

Our primary recommendation is that your exported "artifact" and personal research platform should be one and the same. Using one-off shell commands and later trying to package them up for publishing will yield many untested code paths and invariably bugs that can only be found through actual use. Another tempting approach might be to provide opaque and high-level scripts solely for the purpose of artifact evaluation, which allow reviewers to run a simple command (e.g., "`./run.sh`") and view figures without further input. While it is tempting to use this approach to ease the burden

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike International 4.0 License.

P-RECS '22, June 30, 2022, Minneapolis, MN, USA
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9313-3/22/06.
https://doi.org/10.1145/3526062.3536355

[1] We previously discussed this topic in a post on the SIGOPS blog [4].

on artifact reviewers, we argue that there is value in intermediate results, which help not just artifact reviewers but also system users and the authors themselves gain insights into the system's functionality.

Indeed, publishing the scripts already in use for the project makes them more likely to work for someone else. Of course, given the nature of research, it's not desirable to over-engineer the infrastructure and experimentation scripts. Especially as a graduate student focusing on a single project, it is easy to fill time with infrastructure tasks that feel productive but are not actually advancing your research. While there is no one-size-fits-all approach, a balance between the two is desirable.

For Bundler, we initially wrote special-purpose one-off scripts (with, for example, hard-coded machine IPs and no attempt to manage dependencies) to quickly pull together an initial paper submission. When the paper was initially rejected, we knew from the reviews that the ideas had some promise, but there was still some work to be done in terms of improving the system and designing new experiments, so we created an experiment platform based on knowledge of the types of experiments we would want to run and what the iteration process would look like. We also had fresh in our mind a lot of the mistakes and shortcomings from the initial iteration of the project and recent projects. Incorporating all of this made it much easier to design a clean and robust platform than if we had tried to do it right from the beginning.

Given our experience, we would recommend starting the process of thinking about an experimentation platform *before* the first submission. At the very beginning of a project, we suggest starting with quick iteration, not overthinking organization or scripting, but taking detailed notes about environment setup. Once the project begins to mature, we suggest investing time in creating an experiment platform. This process should not be a burden; a little bit of intentionality will go a long way, and it is not necessary to instrument everything from the beginning. The most important thing is to design it to be easily extensible so that it can grow with the project.

3 Designing Reproducible Experiments

The Bundler [5] system is a middlebox which uses congestion control to "shift" traffic bottlenecks from the middle of the network, where congestion occurs, to a domain with better knowledge of the right scheduling policy. For example, Bundler can ensure that, even in the presence of downstream congestion, latency-sensitive applications will not be delayed by bulk long-lived traffic which keeps network buffers full.

When creating an experimental artifact (or, indeed, evaluation experiments) for either Bundler or any system that depends on congestion control behavior, it's tempting to jump straight into experiments "in the wild". These experiments can demonstrate practical deployability and usefulness: two properties any systems researcher strives to achieve. As a result, program committees tend to especially focus on such experiments.

However, such a "real-world-first" approach has major drawbacks. Firstly, in the Internet, the ground truth conditions are fundamentally unknowable. This makes it hard to reason about a system's performance: did the application perform well because the system worked, or because "Internet weather" cooperated during the experiment? For this reason, "real-Internet" experiments are hard to reproduce and draw useful conclusions from except at truly massive scale (and even then difficult [7]).

Instead, we chose to base our Bundler experimental artifact primarily on experiments in emulated testbed environments, where we controlled both Bundler's behavior as well as that of other network flows. In this environment, we could gather and report ground-truth instrumentation to more deeply demonstrate Bundler's behavior. An example is Figure 10 from our EuroSys paper [5], which would have been impossible to generate outside an emulation testbed. Further, these experiments benefit reproducibility since it is straightforward for the public to run them, unlike a large-scale Internet measurement study would be.

Of course, some "real-Internet" experiments are still necessary, but their role should be to validate the model of the world the emulated environment implements. If a system behaves the same way in an uncontrolled setting as it does in the testbed, this is an indication that the model is useful.

4 Important Components of Bundler Artifact

Now we'll describe four components of the artifact we released [2] for Bunler that help illustrate some of the principles we've described so far. These components were highlighted by the EuroSys'21 AE committee as helpful, but were also crucial to our own research process.

As an overview, we structured our evaluation process as a single evaluation script that takes as input a single configuration file specifying both the system setup and the set of things to do in the experiment. It runs everything, places output files in a specific directory structure, and finally generates an experiment dashboard webpage (details in §4.3). The webpage is the primary point for interacting with the results, but the directory structure makes it easy to go and find specific files for manual inspection as necessary.

The important thing to note is that none of this requires that much work; it is just a small restructuring of the work we had to do in the process of research anyway.

4.1 Monolithic Configuration Files

A hallmark of networking and distributed systems experiments is the complexity of the environment. The results might not just be impacted by code but additionally by any number of small configuration parameters or networking settings on any of the host machines. Thus, ensuring a consistent environment is critical for robustness. Although this is inherent to systems research and cannot be avoided entirely, a little bit of effort goes a long way towards mitigating many common problems.

A common method of configuring experiments is to use command line arguments. While this can work, it encourages specifying a minimum number of parameters, leaving the rest to fall back to default values which can later cause problems. Further, shell history might not include versioning information, so it's unclear which version was run. Finally, carefully specifying all parameters for each invocation quickly becomes annoying and unwieldy.

Instead, our primary recommendation is to represent experiments with a single, verbose configuration file (we use TOML files to help with human-readability) that is explicit about everything. For example, there are some common things that will be important for many projects:

- Platform (GCP, AWS, laptop) and machine type (e.g., `t2-medium`) or hardware details if local
- Exact OS image and kernel version
- Git hash of all relevant repositories (we use a "parent" git repository which includes all experiment dependencies as submodules)
- Sysctls (e.g., setting TCP socket buffer sizes, a problem we had early on)
- OS package dependencies (anything 'apt-get install''d)
- Compiler and hardware driver versions

Tips. A common use case, especially for a graph in a paper, is to do a parameter sweep with the same experiment setting while varying one or a few parameters. This is easily encoded in a configuration file by allowing any values to be either constant OR a list. Then, the experiment script can generate a cross product of all configurations. This makes it easy to, e.g. run all of the experiments necessary for an entire graph in the paper with just one configuration file. See this function[2] in our code for a concrete example.

4.2 Run Experiments with a Single Centralized Script

Especially when working with multiple machines, we recommend a single centralized script that takes this configuration file as input and controls all of the machines, rather than "run script X on machine A, then script Y on machine B" which we have seen as artifact reviewers. This has a few important benefits:

- It easily scales to more machines as you scale up your experiments.
- Building on the configuration files above, the centralized script can be a point of reference to make sure that all machines have the same consistent environment before running. For example, for our artifact we needed a copy of our github repo (which itself had submodules) on each of several machines. At first, we kept running into some variant of the following problem: we would push an update to our repo, then forget to pull it (or forget to update submodules) on one of the machines. The experiment's behavior was wrong, but it did not break completely and was thus hard to debug.
- It can be easier to generalize to different backend platforms. For example, we ran most of our experiments on a local cluster, but by simply changing a few lines in our config experiment files, we could run on Cloudlab.

Specifying everything in a configuration file is only half the battle. For example, in our centralized script, we first check all of the configuration parameters on all of the machines. If anything does not match the config file, the script aborts.

Tips.

- As your script runs experiments, it should first check if a result file already exists where it's expecting to place one. If so, it should stop and ask the user before continuing. This saved us more than once from overwriting some results we were in the middle of investigating that would have been difficult to replicate.
- Add a "dry-run" mode to your script, which simply prints out each command it's going to run without actually running it. We used this constantly for debugging throughout the process of writing our experiment script.

[2] `enumerate_experiments`, at Line 133 of `config.py` in [2]

4.3 Experiment Dashboards

While actually working on a project, experiments are often ad hoc as you quickly iterate and explore different ideas. If you're not careful, it's easy to end up with a mess of a directory structure. It doesn't take long before you forget what's the difference between `exp3/v2/tmp/plot4.png` and `exp3/plot4.jpg`, or where to find the data and script that generated that plot (was it `plot.py` or...`plot2.py`? or `exp3/tmp/plot.py`? Hint: probably none of the

Figure 1: **Sample webpage dashboard for single experiment.**

above!). In order to avoid this mess and keep track of everything, we used experiment dashboards.

Figure 1 contains an example of the dashboard for a single experiment (the configuration file at the bottom was the input to our evaluation script that generated this report). A computational notebook (e.g. jupyter) can also serve as a dashboard.

- Each dashboard is a web page hosted by one of our servers where all team members can easily access it from their browser. This makes it very easy to share plots and data (no sending different versions of plots back and forth across email or slack and having trouble finding them later) and it ensures that everyone has a consistent view of the data.
- A dashboard includes all the plots relevant for the experiment in one place. It also specifies the exact path where the data used to generate the plot is located, so that it's easy to go back and inspect it manually to trace down an issue with the results.
- It also includes the full configuration file used to generate the experiment so that there's no questioning which parameters were used or even which version of the code was used to generate these specific plots. This helps narrow down differences from two experiments that seem otherwise the same (e.g., maybe one piece of code was using a different git hash!)

From the perspective of an artifact reviewer (or a fellow researcher who wants to build on or compare against your system), these dashboards also provide more details than just the figures that made it into the paper, so they can "dig deeper" into the intuition behind the results. This gives them visibility into the intermediate steps rather than the opaque pipeline of running a single script and getting a single static plot at the end.

Details. We generated the interactive plots using plot.ly's R bindings and arranged everything into a webpage using knitr with R-markdown.

4.4 Direct Link Between Experiment Data and Paper

The traditional paper-writing pipeline, where scripts produce a graph which is copied into the paper repository has two problems: (1) it is hard to keep track of how to produce each figure in the paper, and (2) there is no link between the data and in-text reports of results, so when updating a figure it's easy to forget to update sentences such as "In Figure X, our system improves median FCT by Y%".

To avoid these issues, our repository doesn't contain any pre-built graphs, and our paper text doesn't include any hardcoded result numbers. Instead, our Makefile depends upon raw experiment data files. It handles generating figures, parsing important results from those figures, and then defining them as LaTeX variables. This makes it easy to plug in new results and have all dependencies within the paper update automatically. Further, it makes it easy for artifact reviewers (and others) to trace the lineage of a paper figure back to the experiment configuration file that generated it.

Recently, Kassig and Singla have proposed CodeBind [6] to further strengthen the ties beyond knitr's capabilities: in their system, not just the plotting, but the system configuration itself is tied to the paper. This has the potential to combine the benefits of the experiment dashboards we used with the benefits of tying experiment data to the paper.

Details. Similarly to our dashboards, we used knitr to bridge the gap between R (for plotting) and LaTeX. For an example of how this is done, see the references to 'graphs/%.Rnw' in the Makefile of our paper repository [3]. Note that we had some issues with both Overleaf and GitHub not liking large data files, so we hosted them separately.

5 Conclusion

We don't claim to have the answer to the perfect artifact implementation, but we would certainly like to see artifacts improve. Further, we would like to see artifact evaluation become a larger part of the paper review process. Although it would increase the complexity of the review process, it would offer compelling benefits:

- It is a great opportunity for students to get early exposure to the review process and get integrated into the broader research community (similar to shadow PCs).
- We should continue to shift more burden onto the authors to make their artifacts easily reproducible and minimize the amount of work necessary for artifact reviewers. As explained in this post, it can provide a huge benefit to the authors independent of the AE process.

References

[1] 2021. Reproducible Experiments for Useful Internet Systems. https://sysartifacts.github.io/.

[2] Frank Cangialosi and Akshay Narayan. 2021. Bundler EuroSys Artifact. https://github.com/bundler-project/evaluation/.

[3] Frank Cangialosi and Akshay Narayan. 2021. Bundler EuroSys LaTeX Source. https://github.com/bundler-project/writing/.

[4] Frank Cangialosi and Akshay Narayan. 2021. Reproducible Experiments for Useful Internet Systems. https://www.sigops.org/2021/reproducible-experiments-for-useful-internet-systems/.

[5] Frank Cangialosi, Akshay Narayan, Prateesh Goyal, Radhika Mittal, Mohammad Alizadeh, and Hari Balakrishnan. 2021. Site-to-Site Internet Traffic Control. In *EuroSys*.

[6] Simon Kassig and Ankit Singla. 2021. CodeBind: Tying Networking Papers to Their Experiment Code. https://github.com/snkas/codebind-paper.

[7] Bruce Spang, Veronica Hannan, Shravya Kunamalla, Te-Yuan Huang, Nick McKeown, and Ramesh Johari. 2021. Unbiased Experiments in Congested Networks. In *IMC*.