Distributed Systems Labs and Framework

Ellis Michael
University of Washington

DSLabs is a new framework for creating, testing, model checking, visualizing, and debugging distributed systems lab assignments.

The best way to understand distributed systems is by implementing them. And as the old saying goes, "practice doesn't make perfect, perfect practice makes perfect." That is, it's one thing to write code which usually works; it's another thing entirely to write code which works in all cases. The latter endeavor is far more useful for understanding the complexities of the distributed programming model and specific distributed protocols.

Testing distributed systems, however, is notoriously difficult. One thing we found in previous iterations of the distributed systems class at UW is that many students would write implementations which passed all of our automated tests but nevertheless were incorrect, often in non-trivial ways. Some of these bugs would only manifest themselves in later assignments, while others would go entirely unnoticed by our tests. We were able to manually inspect students' submissions and uncover some of these errors, but this approach to grading does not scale and does not provide the immediate feedback of automated tests.

The DSLabs framework and labs are engineered around the goal of helping students understand and correctly implement distributed systems. The framework provides a suite of tools for creating automated tests, including model checking tests which systematically explore the state-space of students' implementations. These tests are much more likely to catch many common distributed systems bugs, especially bugs which rely on precise orderings of messages. Moreover, when a bug is found, these search-based tests output a trace which generates the error, making debugging dramatically simpler. Finally, DSLabs is integrated with a visual debugging tool, which allows students to graphically explore executions of their systems and visualize invariant-violating traces found by the model-checker.

Programming Model

The DSLabs framework is built around message-passing state machines (also known as I/O automata or distributed actors), which we call *nodes*. These basic units of a distributed system consist of a set of message and timer handlers; these handlers define how the node updates its internal state, sends messages, and sets timers in response to an incoming message or timer. These nodes are run in single-threaded event loops, which take messages from network and timers from the node's timer queue and call the node's handlers for those events.

This model of computation is typically the one we use when introduce distributed systems for the first time and the one we use when we want to reason about distributed protocols and prove their correctness. The philosophy behind this framework is that by creating for students a programming environment which mirrors the mathematical model distributed protocols are described in, we put them on the best footing to be able to reason about their own implementations.

Testing and Model Checking

The lab infrastructure has a suite of tools for creating automated test cases for distributed systems. These tools make it easy to express the scenarios the system should be tested against (e.g., varying client workloads, network conditions, failure patterns, etc.) and then run students' implementations on an emulated network (it is also possible to replace the emulated network interface with an interface to the actual network).

While executing certain scenarios is useful in uncovering bugs in students' implementations, it is difficult to test all possible scenarios that might occur. Moreover, once these tests uncover a problem, it is a challenge to discover its root cause. Because the DSLabs framework has its node-centric view of distributed computation, it enables a more thorough form of testing – *model checking*.

Model checking a distributed system is conceptually simple. First, the initial state of the system is configured. Then, we say that one state of the system, s_2, (consisting of the internal state of all nodes, the state of their timer queues, and the state of the network) is the successor of another state s_1 if it can be obtained from s_1 by delivering a single message or timer that is pending in s_1. A state might have multiple successor states. Model checking is the systematic exploration of this state graph, the simplest approach being breadth-first search. The DSLabs model-checker lets us define *invariants* that should be preserved (e.g. linearizability) and then search though all possible ordering of events to make sure those invariants are preserved in students' implementations. When an invariant violation is found, the model-checker can produce a *minimal trace* which leads to the invariant violation.

While model checking distributed systems is useful and has been used extensively in industry and academia to find bugs in distributed systems, exploration of the state graph is still a fundamentally hard problem – the size of the graph is typically exponential as a function of depth. To extend the usefulness of model checking even further, the test infrastructure lets us prune the portion of the state graph we explore for an individual test, guiding the search towards common problems while still exploring all possible executions in the remaining portion of the state space.

The DSLabs model is built to be usable by students and be as *transparent as is practical*. Students will be required to make certain accommodations for the model checker's sake, but we try to limit these and provide tools that help validate the model checker's assumptions and debug model checking performance issues. Moreover, the model checker itself is not designed with state-of-the-art performance as its only goal. Building a model checker that can test student implementations of runnable systems built in a general-purpose language such as Java requires striking a balance between usability and performance.

Visualization

This framework is integrated with a visual debugger. This tool allows students to interactively explore executions of the distributed systems they build. By exploring executions of their distributed system, students can very quickly test their own hypotheses about how their nodes should behave, helping them discover bugs in their protocols and gain a deeper understanding for the way their systems work. Additionally, the tool is used to visualize the invariant-violating traces produced by the model-checker.

Assignments

We currently have four individual assignments in this framework. In these projects, students incrementally build a distributed, fault-tolerant, sharded, transactional key/value store! - Lab 0 provides a simple ping protocol as an example. - Lab 1 has students implement an exactly-once RPC protocol on top of an asynchronous network. They re-use the pieces they build in lab 1 in later labs. - Lab 2 introduces students to fault-tolerance by having them implement a primary-backup protocol. - Lab 3 asks students to take the lessons learned in lab 2 and implement Paxos. - Lab 4 has students build a sharded key/value store out of multiple replica groups, each of which uses Paxos internally for replication. They finish by implementing a two-phase commit protocol to handle multi-key updates.

Parts of this sequence of assignments (especially labs 2 and 4) are adapted from the MIT 6.824 Labs. The finished product is a system whose core design is very similar to production storage systems like Google's Spanner.

We have used the DSLabs framework and assignments in distributed systems classes at the University of Washington.

Directory Overview

- `framework/src` contains the interface students program against.

- `framework/tst` contains in the testing infrastructure.
- `framework/tst-self` contains the tests for the interface and testing infrastructure.
- `labs` contains a subdirectory for each lab. The lab directories each have a `src` directory initialized with skeleton code where students write their implementations, as well as a `tst` directory containing the tests for that lab.
- `handout-files` contains files to be directly copied into the student handout, including the main `README` and `run-tests.py`.
- `grading` contains scripts created by previous TAs for the course to batch grade submissions.
- `www` contains the DSLabs website which is built with Jekyll.

The `master` branch of this repository is not setup to be distributed to students as-is. The `Makefile` has targets to build the `handout` directory and `handout.tar.gz`, which contain a single JAR with the compiled framework, testing infrastructure, and all dependencies. The `handout` branch of this repository is an auto-built version of the handout.

Contributing

The main tools for development are the same as the students' dependencies — Java 17 and Python 3. You will also need a few utilities such as `wget` to build with the provided `Makefile`; MacOS users will need `gtar` and `gcp` provided by the `coreutils` Homebrew package, and `gsed` provided by its own Homebrew package.

IntelliJ files are provided and include a code style used by this project. In order to provide IntelliJ with all of the necessary libraries, you must run `make dependencies` once after cloning the repository and whenever you add to or modify the project's dependencies. You will also need the Lombok IntelliJ plugin.

This project uses `google-java-format` to format Java files. You should run `make format` before committing changes. If you want IntelliJ to apply the same formatting, you will need the `google-java-format` IntelliJ plugin, and you will need to apply the necessary post-install settings in IntelliJ.

If you add fields to any student-visible classes (all classes in the `framework` package as well as `SearchState` and related classes), you should take care to ensure that `toString` prints the correct information and that the classes are cloned correctly. See `dslabs.framework.testing.utils.Cloning` for more details. Also see Lombok's `@ToString` annotation for more information about customizing its behavior. In particular, note that **`transient`** and **`static`** fields are ignored by default by all cloning, serialization, and `toString` methods.

Acknowledgements

The framework and labs have been improved thanks to valuable contributions from: - Alex Saveau - Andrew Wei - Arman Mohammed - Doug Woos - Guangda Sun - James Wilcox - John Depaszthory - Kaelin Laundry - Logan Gnanapragasam - Nick Anderson - Paul Yau - Sarang Joshi - Thomas Anderson

The lab assignments, especially labs 2 and 4, were adapted with permission from the MIT 6.824 labs developed by Robert Morris and colleagues.

Contact

Bug reports and feature requests should be submitted using the GitHub issues tool. Email Ellis Michael (emichael@cs.washington.edu) with any other questions.

If you use these labs in a course you teach, I'd love to hear from you!