Testing Complex Systems

An overview of a layered approach to testing complex systems.

June 10, 2014
URL:http://www.drdobbs.com/testing/testing-complex-systems/240168390

Testing large distributed systems is hard. There is so much more you need to test beyond the exercising code with unit tests. Performance testing, load testing, and error testing must all be undertaken with realistic usage patterns and extreme loads. But exhaustive tests are often slow to run and hard to maintain; and over-fitting the tests to the code can make it difficult to refactor the code. So many problems! In this article, I discuss testing complex systems using a layered approach that is both manageable and delivers comprehensive coverage.

Big distributed systems can't be fully tested on one developer machine. There are several levels of testing stretching over a range of speeds, resources, and fidelity to a production system. For example, a typical large system might consist of thousands of various servers — front-end Web applications, REST API servers, internal services, caching systems, and various databases. Such a system might processes several terabytes of data every day and its storage is measured in petabytes. It is constantly hit by countless clients and users. It is kind of difficult to replicate all this on a developer's notebook.

Layered Testing

The key to tackling testing on complex systems is layered testing in special environments. The basic layer is composed purely of unit tests using mocks. This practice is well-defined and I will not spend much time talking about it (except for the limitations and dangers of excessive mocking). Unit tests can be run by individual developers on individual machines and should run quickly. How often the tests are run is up to the developer, but at the very minimum, all the tests should be run once before check-in. It is common to run a subset of tests that target the specific code the developer works on very frequently (every few minutes).

The next layer, which can still be run on a developer machine, is integration testing using virtual machines (VMs) and/or other isolation techniques, such as Linux containers, to simulate a production system or some aspects of it. For example, you can deploy a database on one VM and API server on another, then have them talk to each other in the same way they do in production.

This layer builds confidence that the code does not contain incorrect assumptions about various components of the system running on the same machine. It also allows full workflow in a configured, multiple (virtual) machine setting. A developer will run these tests at least once before checking-in any code, and more frequently if working on code that is not exercised fully by unit tests.

The next layer is integration tests + system tests running in a shared environment. This often involves the continuous integration/auto-build environment. The main purpose of this layer is to make sure that recent code checked in by a developer doesn't break anything obvious that will prevent other developers from moving forward. There may be also a separate development environment were developers can run various system-level experiments. These tests should be run automatically whenever a developer checks in any code, as there are sometimes complex issues related to source control practices, batched changes, and handling merges.

The next layer is composed of performance tests, load tests, disaster recovery tests, backup-restore tests, rolling update tests, and other system-wide tests. These tests are typically performed in a staging environment, which should be as close as possible to the production environment. The frequency of these tests is determined on a case-by-case basis. They could be run daily or weekly. At the very least, they should be run before preparing a release candidate, and the results examined carefully and compared to previous runs.

The final layer of tests are production tests. These tests are typically performed on large-scale systems that already have mechanisms for dealing with partial outages, and are believed to be robust to localized or partial failures. A stellar example of production-level tests can be found in the Netflix simian army. It is a collection of programs that randomly cause various failures of different magnitudes, so that it's possible to make sure the system under test can handle them. Not many companies reach this level of testing.

Environments

The purpose of an "environment" is to support isolated and repeatable deployment of a known configuration of the system. This includes a particular version of each component, third-party library, configuration file, set of users, permissions, etc. This is particularly important between staging- and production-level testing.

So, how do you setup and maintain an environment? The best practice for creating and managing environments is to use a configuration and orchestration tool. In the past, this was traditionally done with some combination of shell scripts, in-house systems, a lot of manually babysat servers, praying, and cursing. In recent years, as deployments have increased significantly in size, this approach couldn't scale. A slew of tools like Chef, Puppet, SaltStack, and Ansible have arisen to address the needs of current software builds. These tools enable developers to more reasonably manage the entire "environment" infrastructure of machines, configurations, and code deployments.

I use Ansible, which is implemented in Python but uses a set of declarative YAML files to specify the state a particular machine should be in. This design allows repeatable provisioning and configuration of multiple machines. Specifically, Ansible uses playbooks and inventory files. The playbooks specify what needs to be done on each machine, and the inventory describes what playbooks or roles should be applied to each specific machine or group of machines.

Local environments can be created and destroyed at will. Shared environments that are used by multiple developers and often comprise multiple physical and/or virtual machines and must be managed more carefully. Tests on shared environments should either be serialized (one test runs at a time and the environment is restored to its initial state after each test), or the tests should be separated (for example, isolating all artifacts and changes from a particular test to a timestamp-based directory or namespace). The serialized approach is safer and easier to implement, but if integration and system tests take a long time, it can delay development. The approach of a separate test suite is harder to implement correctly and not as safe, because side-effects from one test may affect another test. This is almost unavoidable when tests need to use a common scarce resource (like a database) that can't be replicated for each test.

Note that when multiple developers use a shared environment to run tests, each developer needs to run the entire suite of tests.

There are no easy off-the-shelf answers here. You will have to tailor your process to your situation, and walk the fine line between more rigorous but slow tests and faster but less rigorous tests.

Mock Objects

Mock objects are great for testing complex systems. They typically stand in for some dependency that your code calls. The mock object usually has some canned response that mimics the original dependency in a given situation. But when testing distributed systems, you often want to test against a certain input stream of requests, messages, events, or user actions. The concept is similar: You want to replace the real actor with something you control. In unit tests, when you test a single method, you simply provide the input arguments as part of the test. When testing a distributed system over time and multiple transactions, you need something else. In such cases, I have found it useful to write a `Simulator` class that generates the message and data streams that can then percolate through the system.

Mock objects have some important limitations:

- It is difficult to mock objects that return complex data structures: Consider an object that returns a complex graph of objects that have various dependencies and relationships. It is very tedious, error-prone, and labor-intensive to construct the canned response necessary for mocking.
- Mocks can get out of sync with the object they mock: This is particularly nasty in error-handling situations. Suppose a method used to return an error code when a certain operation failed, but the behavior changed to raise an exception. If the object that calls the method is still expecting an error code and the mock object wasn't updated to raise an exception, then the test will still pass. So, the code will be deployed to production and can run for years until this error condition happens — then all hell will break loose.

Testing Web Applications, Services, and APIs

In the Python world, there are many solid and battle-tested Web frameworks. They cover the entire spectrum from a behemoth like Django to a single file micro-framework like Bottle. Many of them provide extensions for publishing REST APIs. Here is quick an example of tests built on top of the Flask framework and the Flask-RESTful extension.

Fully testing Web applications is often a difficult task. Sophisticated interfaces can often be tested properly only by humans. Web services that return only data (JSON, YAML, csv) are easier. Flask provides good support for controller testing via an internal test client and its development server. You can read more about Flask and check out some tests for a play server I wrote for details and examples.

HTTP is the transport of choice for many service-oriented distributed systems. As such, there are several libraries that capture HTTP-based interactions and can replay them later. The vcrpy is a good option for Python.

Testing Data Store Code

Any non-trivial system will have a persistent layer. It could be a relational database, it could be a NoSQL database, it could be some cloud storage, and more than likely, it consists of some combination of these For many reasons (caching, security, flexibility, and testing), it is recommended that you hide your actual data stores behind a data-access layer. If you do this, then you should be able to mock your data store for most unit and integration tests. However, sometimes you do need to exercise your data store code, too. This will happen in system-wide tests, and also when you have a relational database in the mix (or something like MongoDB) and there is a lot of business logic associated with your data model. In these cases, you will want to have a test data store that you can populate with test data (more on how to do that later). For relational databases, Python has a great option in the form of SQLite, which comes with Python since version 2.5. SQLite is an embedded database that's super fast and can even be instantiated in memory. I have used it successfully to test a lot of data-oriented code.

Some parts of advanced systems are not controlled by code, but rather, are data driven. The code consists of some generic workflow and the actual logic and the specific operations that take place are derived from external data stored in configuration files, databases, or even a remote service. This kind of system, while very flexible and powerful, is harder to maintain and test. Changes in the behavior of the system are not fully reflected by the latest code in source control, but a combination of code and data.

Again, you have to be diligent here and make sure that the data-driven code behaves the same across all environments. Ideally, you will develop specialized tools to manage this process.

Testing Data-Intensive Systems

Beyond unit and integration tests, there are system tests. Large distributed systems typically store and manage lots of data. To test a complicated system properly, you want your test to run on data that has similar properties to production data (data statistics, usage patterns, value range, etc.). There are two common approaches for generating test data.

- Synthetic Data: Artificial data generated by a program that satisfies some desirable properties for the test. It could contain invalid data to see how the

system handles it, or some data for rare use cases that don't show up often in the wild. The benefit of synthetic data is that you have full control. The downside is that it's difficult to maintain, and sometimes it's not clear if it reflects properly production data.
- Production Data Snapshot: With a production snapshot (often from a backup), you know you're running your test on real-world data. The pros and cons are exactly the opposite of the synthetic data approach. You get high-fidelity data, but it's hard to control and test special cases.

It is also possible to combine the two approaches and have a base snapshot of production data, then add some synthetic data for specific use cases.

Dealing with Deployments, Versions, and Upgrades

You're about to roll out a new version of your system, which includes significant changes. You did your part on the testing front from local tests, through integration tests, and all the way to various system tests in the staging environment. But it doesn't matter how good your tests are. Unless you have an exact replica of your production environment for testing, you can't be totally sure what the impact going to be. Obviously, you can't risk breaking your production environment, corrupting your data, or making it unavailable or too slow. There are several strategies that will let you test the water.

First and foremost, you should be able to quickly revert any code and configuration changes at the first sign of trouble. But some changes go beyond the code (particularly in heavily data-driven systems). A good strategy in such cases is to do a multi-phase deployment. First, deploy the new changes side-by-side with the existing system. Whatever new data you collect will be stored in both the "old" way and the "new" way. Accessing data will still be done via the "old" way. Now you have new data accumulating both ways, in parallel, and you can verify everything is working well by comparing old to new. Once you are sure everything is OK, you can roll out another change that switches the data access to the new data and move entirely to the "new" way. You still collect data using the "old" way, too, so if something is wrong, you can just roll back the data access change and no data will be lost or corrupted. Finally, after the new data is accessed, you can phase out the "old" way of storing the data (which may involve migration).

Another method for controlled rollout of new changes is to deploy to a small number of servers first. If there is any problem, roll back and fix. This, of course, requires that the distributed system can work with both old and new nodes at the same time.

Separating Production Code from Test Code

Building testable systems is hard. You must design for testability or at least refactor towards testability. You're in bad shape if your code is littered with lots of conditionals like:

```
if TEST_MODE:
 do_test_thingy()
else:
 do_production_thingy()
```

You may think it's OK because there is just one global switch `TEST_MODE` and it is easy to make sure it is always `False` in production. But the reason it is really bad is that it means that `do_test_thingy()` is callable from your production code and can be be executed unconditionally somewhere else. Your goal should be to eliminate as many issues as possible ahead of time.

Note that it may be workable to have a single switch to select a test configuration file (or something similar) to control the entire behavior in one place. It is often a good idea to have the switch be an environment variable that contains the name of the configuration file:

```
config_file = os.environ.get('ACME_CONFIG_FILE', 'config.py')
```

Another bad habit is having various test utilities that may be handy for non-test scenarios. If the boundary is vague, people will use a cool test decorator that retries every operation five times in their production code to protect against slow response from some service. Next thing you know, someone decides to arbitrarily change some knob in the retry decorator to make a test pass (because it's only a test decorator), and suddenly your system hangs for 10 minutes when a server is down instead of failing fast. Mixing production and test code is a slippery slope. The relationship should be unidirectional: Tests use production code. If you have shared code that you want to use in both tests and production code, move that code to a separate non-test module or package. Then, it can be used by anyone.

The best practice is: Production code never imports any test code. If multiple tests from multiple packages want to share some test helpers, put this shared test code in a separate test helpers package that only tests will import and use.

Conclusion: Be Reasonable

"Be Reasonable." This seems like good advice in general. As you no doubt realize, proper testing is hard. Typically, you won't be able to fully test everything. If you decide to focus on testing and really get as close as possible to the holy grail of a fully tested system, you may find that you're reaching diminishing returns. It may be very difficult to evolve the system and make changes. Exhaustive tests may slow you down. The tests, themselves, may become so complicated that they will contain their own bugs, and when a test fails, you may have a hard time seeing what the test is complaining about whether it really is a failure.

When you have layered code, consider skipping testing each and every layer. Maybe testing from the outside is good enough. Match your testing level to the criticality of the subsystem under test. Take into account other factors like how easy it is to recover from failures and how easy it is to fix them with and without tests.

Even with all this thoughtful reasoning, I can still assure you that you don't test enough.

Gigi Sayfan specializes in cross-platform object-oriented programming in C/C++/ C#/Python/Java with emphasis on large-scale distributed systems.