DOI:10.1145/2776756

Article development led by acmqueue
queue.acm.org

Testing a distributed system can be trying even under the best of circumstances.

BY PHILIP MADDOX

Testing a Distributed System

DISTRIBUTED SYSTEMS CAN be especially difficult to program for a variety of reasons. They can be difficult to design, difficult to manage, and, above all, difficult to test. Testing a normal system can be trying even under the best of circumstances, and no matter how diligent the tester is, bugs can still get through. Now take all of the standard issues and multiply them by multiple processes written in multiple languages running on multiple boxes that could potentially all be on different operating systems, and there is potential for a real disaster.

Individual component testing, usually done via automated test suites, certainly helps by verifying that each component is working correctly. Component testing, however, usually does not fully test all of the bits of a distributed system. Testers need to be able to verify that data at one end of a distributed system makes its way to all of the other parts of the system and, perhaps more importantly, is visible to the various components of the distributed system in a manner that meets the consistency requirements of the system as a whole.

This article discusses general strategies for testing distributed systems as well as specific strategies for testing distributed data storage systems.

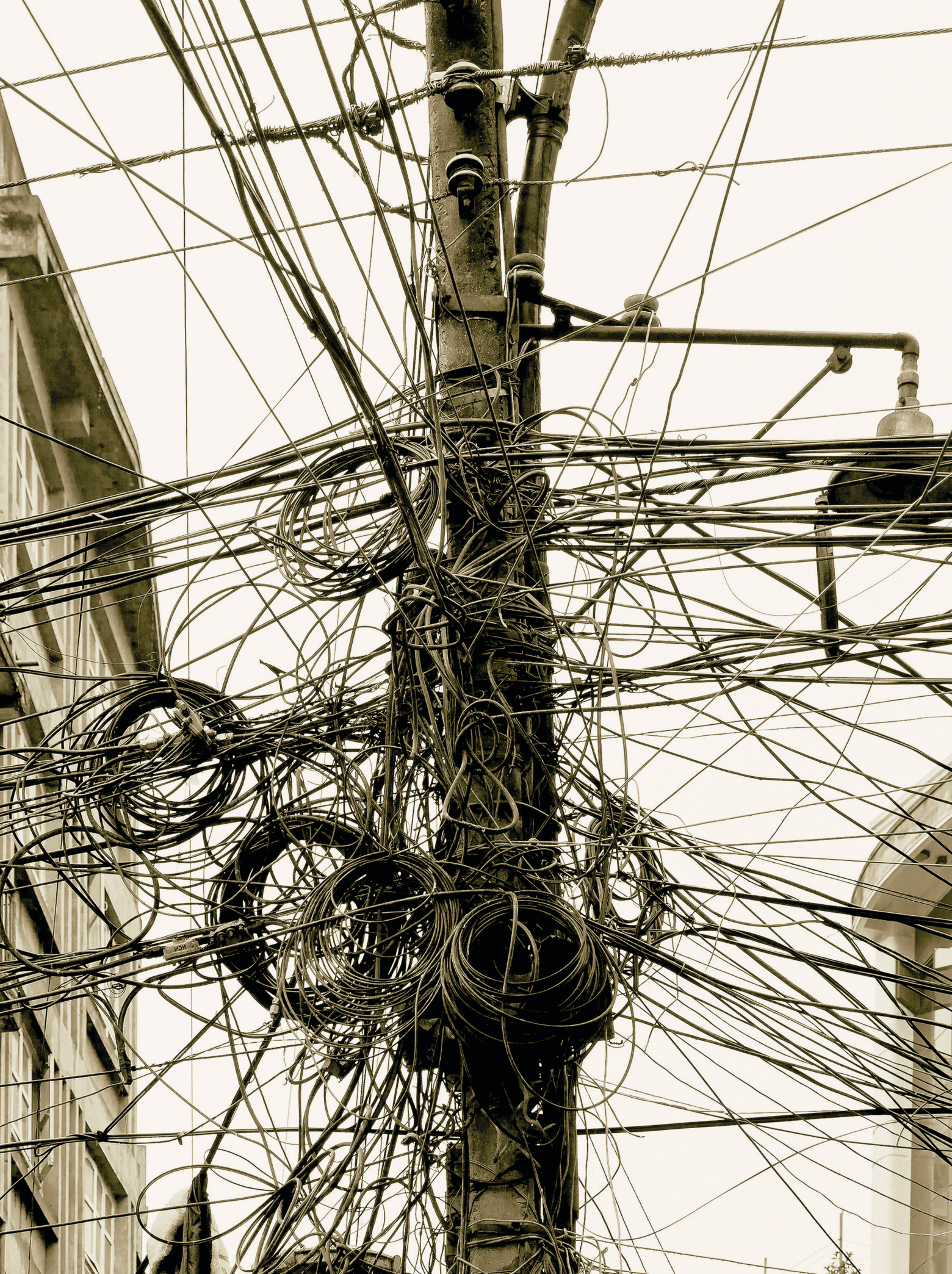
End-to-End Testing

A common pitfall in testing is to check input and output from only a single system. A good chunk of the time this will work to check basic system functionality, but if that data is going to be propagated to multiple parts of a distributed system, it is easy to overlook issues that could cause a lot of problems later.

Consider a system with three parts (illustrated in Figure 1): one that collects data, one that collates all of the data into consistent messages, and one that receives the data and stores it for later retrieval. Test suites can be easily written for each of these components, but a lot can still go wrong from one end to the other. These errors can be tricky—issues such as order of data arrival and data arrival timing can cause all sorts of bugs, and there are so many possible intersections of these things throughout the system that it is difficult to predict and test all of them.

A good way of addressing this problem is to make all the components of the system configurable so they can be run locally. If all of the components of a system can run on the same box, local end-to-end tests are effective. They provide control over when instances of the system elements are brought up and shut down, which is difficult when the elements are spread across different machines. If it is not possible to co-locate processes on the same box, you can run them in virtual machines, which provides much of the same functionality as running them on the same machine. An effective way of testing is to write a series of individual component tests that verify each component is working properly, then write a series of tests that verify data delivery is working properly from one end of the system to the other.

IMAGE BY VADIM PETRAKOV



The example given in Figure 1 would not be difficult to test—after all, there are only three working parts. In the real world, however, highly distributed systems can have hundreds of individual components to run. In addition, most real-world distributed systems are not simply chains of systems that cleanly execute from one end to the other. To demonstrate this point, let's expand the previous example to include hundreds of data collectors throughout the world, each pumping data into the collator, which then pushes all of the data into a distributed data storage system with a configurable number of storage nodes (as shown in Figure 2).

It is nearly impossible to test all the configurations. Typically, the best that can be done is to come up with a good approximation of how this system will work, using a variable number of data collectors and a variable number of storage nodes.

You may also encounter scenarios in which you have access to only a single component of a distributed system. For example, you could be writing a third-party system that collates data from a variety of different sources to assemble data for a user. More than likely, you do not actually have edit access to the other systems and cannot easily control the output from them. In a situation like this, it is useful to write simulators to mimic various types of output from the other systems, both good and bad. This approach has a few drawbacks:

- It is impossible to know all of the different types of good and bad output the other system may provide.
- Writing various simulators can be time consuming.
- The simulators will not react to receiving data the same way the actual system will.

Simulators can be valuable tools to help develop and debug system components, but be aware of their limitations and do not rely on them too heavily.

Distributed Data Systems

Testing a distributed data store is complicated on its own, even without having to test it on the end of another distributed system. These systems require detailed tests of their own to ensure everything works properly. Sticking them at the end of a distributed-system test can add huge amounts of complexity and time to any test suite. Writing a separate set of tests exclusively for the data store is a good idea; otherwise, the amount of time required to get the data from one end to the other can be frustrating—waiting 30 minutes for data to trickle all the way through a system, only for it to fail a single test, requiring the whole thing to be run again.

While it is still important to test the entire system from beginning to end, a complicated subsystem such as a distributed data store demands to be tested on its own in addition to the full system tests. Distributed data systems have many issues that are very difficult to deal with. Among the most difficult to deal with are asynchronous data delivery and node failure.

Note the eventual consistency requirements for your system may be different than the system described here. In my examples, I am making the following assumptions:

- Data is stored on a certain number of nodes defined when the system is built and the node a certain piece of data is on will not change.
- Data will be sent to a single node, then propagated to the other nodes that need to store copies of the data.
- The system will always return the most up-to-date version of the data, regardless of which node it is stored on.
- If one or more nodes are down, the data will correctly flow to the nodes that were offline once they are brought back online.

Asynchronous Data Delivery

In the custom distributed data storage system that I work on, data is delivered to any one node in the system, which then sends that data asynchronously to all other nodes in the system that are responsible for storing the data. The system that sends the data to the first node does not receive any indication the data has been successfully replicated throughout the system, only the first node received the data properly. The system deals with large enough volumes of data that waiting for synchronous data delivery throughout the entire cluster would be unacceptably slow. This system has worked well for many years, but testing it often takes at least as much time as new development efforts, if not more.

When data is added to the system, testing must verify the data reaches not only the intended node, but also all nodes required to store the data to make the system achieve eventual consistency. Therefore, the test suite needs to pull the data from each node where the data supposedly lives, and not just use a generalized query aimed at the cluster. Without the knowledge of when (or if) the data was propagated properly throughout the rest of the system, however, there is a chance the data has not yet reached its intended destination. It would be possible to add a flag to the executable that would wait until all data has propagated throughout the cluster before returning a valid status to the system adding the data to the cluster. This would guarantee the data has propagated before it is pulled, but it would also alter the manner in which the system works on a

Figure 1. Simple distributed system.

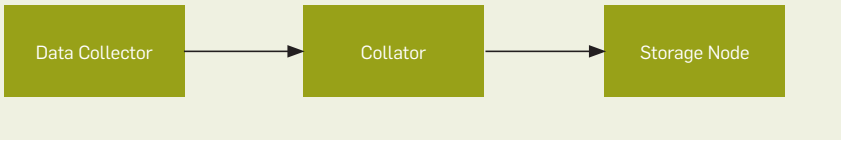
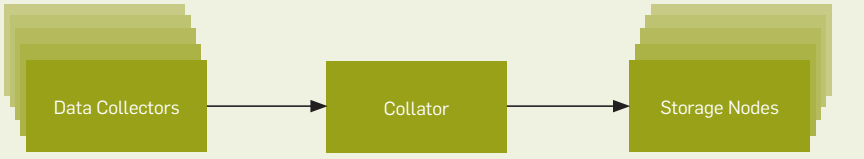


Figure 2. More complex distributed system.



day-to-day basis, and the test would no longer truly be testing normal system functionality.

A solution that maintains normal system functionality is to add artificial delays to the test suite after each successful set of inputs to the primary node. This gives the data time to propagate and increases the chances the data will be on all of the proper nodes by the time an attempt is made to pull it. Obviously, this is not an ideal solution. First, since there are absolutely no timing guarantees, data delivery cannot be guaranteed in any sort of time frame. The timing would also be different based on the speed of the system on which the test suite runs; what works well on one system could be inadequate on another system.

Differences in loads and speeds between the development environment and official build box can also cause problems. I have run a test suite successfully many times on my own development box, only to have the automated build fail the test suite as a result of speed and load differences. This can lead to frustration in trying to determine if there is a legitimate issue with the system or a problem with the test suite.

The obvious solution to this is to increase the number of artificial delays until the system works properly no matter where it runs. These delays, in aggregate, can cause the time the test suite takes to run to increase exponentially. The test suite I use is tied to an automated Jenkins build that runs every time code is committed to our repository. The system is sufficiently complex to necessitate a long, detailed test suite to hit all of the use cases. When enough artificial delays get added to the test suite, the time it takes to run can begin to bloat quickly. A test suite that should take about five minutes to run could end up taking 30 minutes. As more tests are added to the system, it could bloat even more, until running the test suite takes so frustratingly long that you start to ignore it or skimp on adding later tests, leading to bugs slipping through the cracks.

The best way to counteract the impact of this bloating is to run as many tests in parallel as possible. If everything runs in sequence, the time required to run the tests is going to increase rapidly. While the speed of data input likely will not increase if the system is already extremely loaded down, reads can easily be done in parallel.

Testing a distributed data store is complicated on its own, even without having to test it on the end of another distributed system. These systems require detailed tests of their own to ensure everything works properly.

This has an obvious downside—doing your tests asynchronously is much more difficult than simply doing them in sequence. There are far more opportunities for bugs to show up in your test suite itself. You will need to weigh the difficulty of asynchronous testing against the time issues that can crop up if you test everything sequentially. A "hybrid" solution may work well for you—group your tests into small sets, then run the sets sequentially, doing as much of each set in parallel as possible. This allows for minimizing the amount of asynchronous programming you need to do for your test suite while still keeping it reasonably fast.

It is also important to tailor your tests for the requirements of your system. In my example, I am making the assumption there is no specific timing requirement for your system and you can simply extend the amount of time needed to run the tests. However, if you do have specific timing requirements, you cannot simply increase the time required until the tests pass. You will also be dealing with increased loads depending on the number and type of tests being run in parallel. It may become necessary to measure the load and ensure you are reaching your timing requirements based on the amount of data being processed. These are important things to keep in mind when designing your tests, determining the timing in your tests, and determining the amount of load to place on your system when running the test suite.

In the end, there is no ideal way of testing asynchronous data delivery, as each approach has flaws. It really comes down to what works best for a particular system. I find system delays after each successful input to a primary node, combined with high levels of parallel test running, to be the best solution because it most accurately mimics a system actually running in production, without introducing synchronous messages that do not match what the system is doing in production. This is not a one-size-fits-all situation, however. Every distributed data system

is different, and it may make perfect sense for you to add synchronization in testing. Just confirm you are making choices based on what approach best tests your system, not what is easiest for you. When you start writing tests based on ease rather than making sure everything is tested, you are going to start letting bugs through.

Node Failure in a Distributed Data Store

Component failure in a distributed system is difficult to test for. When a component fails, it cannot cause the entire system to fail. The other parts of the system must continue to work properly, even without one or more components. In a distributed data store, ideal behavior is for the node failure to be completely invisible to users. The following situations must be tested to ensure the storage cluster behaves properly:

- Data sent to the system while the node is down still needs to propagate properly throughout the system.
- Data should still be retrievable even with a node failing.
- When the node comes back online, data that must be stored on that node should be propagated to the node and be retrievable from it.
- If the node failure is permanent, the node must be recoverable using data stored throughout the rest of the cluster.

These tests can be via an automated test suite or by setting up a test environment and manually running tests against it to ensure everything is working properly. Once the system is set up and data is flowing into it, you can take a node offline and verify all the data appears to be working. Data can be pulled manually from the data store to ensure it is still retrievable. Once this is verified, the downed node can be brought back online. The data that belongs on this node should begin to flow into the node. After a while, data can be pulled manually from this node to ensure the data that was sent to the cluster when the node was down is stored correctly.

While testing manually is certainly easier, an automated test suite is preferable. As previously discussed, if you can configure your distributed data store to run on one box, you should be able to write an automated test suite to help test it.

The following items need to be guaranteed when testing for node failure in a test suite:

- The test data must have been added to the system and fully propagated throughout the cluster. Several strategies for ensuring this were described earlier.
- When a node is brought down in the test suite, you need to be able to verify it has actually gone down and is inaccessible. A node that did not respond to your signal should not be able to serve data and obscure your results.
- You should be able to pull every single piece of data stored in your distributed data storage system without error. If any pulls fail, then the system has a data-distribution bug that needs to be corrected.

Testing for node failure requires the ability to pull data from individual nodes in the system. It is well worth the time to ensure when attempting to pull data from a node, you can tell the node to do one of the following:

- Provide the "best data" stored in the cluster, along with an indication of where the data came from.
- Provide the data owned by that particular node, regardless of whether or not "better" data is stored elsewhere.

The ability to verify data storage on individual nodes is essential, even if the standard use case is to provide the freshest data. This will help squash lots of bugs before they manifest in the production environment. This may require adding data entry points in the API that are primary (or indeed, exclusively) used for testing, since during typical use, you will likely not care which node the data is served from as long as the data is correct.

Once you have the ability to verify data storage on each individual node in place, it is time to begin testing. The first step is to decide on a node to kill and bring it down. Try to pull data from the node after it fails; the retrieval should fail and exit cleanly. If the retrieval gives back data or hangs, then you need to ensure the data node handles shutdowns properly.

The next step is to pull data from the cluster. You should be able to pull every single data point you added to the cluster, even though a node is down. If any data is missing, you have a data-distribution bug, or the data was not given adequate time to complete all of its asynchronous data distribution before the node was shut down. If the test results are inconsistent from run to run, it is likely the result of asynchronous data-distribution failure. Fix that issue and try again. If the results are consistent, it is much more likely that you have a bug.

If data is stored redundantly across the cluster, then reconstructing a node that has completely failed should be possible. Testing this is similar to testing a node that is down: turn off the node, remove all of the data, rebuild the node, turn the node back on, and verify the data you expect to be on that node actually exists on it. You really want to have this functionality, and you definitely must ensure it works properly. When a production node crashes and loses all of its data, you need to be able to re-create the node quickly and with confidence it is working.

Conclusion

Verifying your entire system works together and can recover cleanly from failure conditions is extremely important. The strategies outlined in this article should help in testing systems more effectively.

It is important to note, however, that no two distributed systems are the same, and what works in some may not work in others. Keep in mind the way your own system works and ensure you are testing it in a manner that makes sense, given the particulars of your system. C

Related articles on queue.acm.org

The Antifragile Organization
Ariel Tseitlin
http://queue.acm.org/detail.cfm?id=2499552

Distributed Development Lessons Learned
Michael Turnlund
http://queue.acm.org/detail.cfm?id=966801

Lessons from the Floor
Daniel Rogers
http://queue.acm.org/detail.cfm?id=1113334

Philip Maddox is a systems engineer at Circonus, a leading provider of monitoring and analytics for IT Operations and DevOps, where he works on a large distributed data storage system. Previously, he wrote code for mail-sorting machines for the U.S. Postal Service.