The University of Texas at Austin

Software Testing

Software Engineering - Option III

Testing Distributed Systems - Kafka

Authors:
Howie Benefiel
Dale Pedinski

Professor:
Dr. Sarfraz Khurshid

August 15, 2018

Abstract

1 Prior Work

From our literature review, we found that the field of testing distributed systems is relatively unexplored. This is backed up by the findings of Yuan et al. [2].

1.1 Simple Testing Can Prevent Most Critical Failures

The authors of Yuan et al. analyzed a number of open-source distributed systems running production workloads in industry currently. Their findings showed that a large percentage of catastrophic failures could have been prevented by simple tests and static analysis.

In Yuan et al.'s paper, they studied the catastrophic failures 5 distributed, data-intensive systems, Cassandra, HBase, Hadoop Distributed File System (HDFS), Hadoop MapReduce, and Redis. They defined a catastrophic failure as a failure which would result in unavailability of the system or data loss. To determine the faults of the system, the authors analyzed failures reported on the respective system's bug tracker. For each system, the authors analyzed the catastrophic failures.

In general, Yuan et al. found that the failure modes of distributed systems were complex. The authors determined that the complexity of these failure modes was caused by the relatively large state space of distributed systems. Quantitatively, they found that 77% of failures required more than one event to produce the failure, and 90% required no more than three.

The authors of Yuan et al. then analyzed the types of inputs required to produce the failure. The most unexpected finding from this analysis is that 24% of failures were caused by a node becoming unreachable. This is surprising because a core tenant of distributed systems is fault-tolerance, i.e., the system should be able to handle node*s* becoming unavailable.

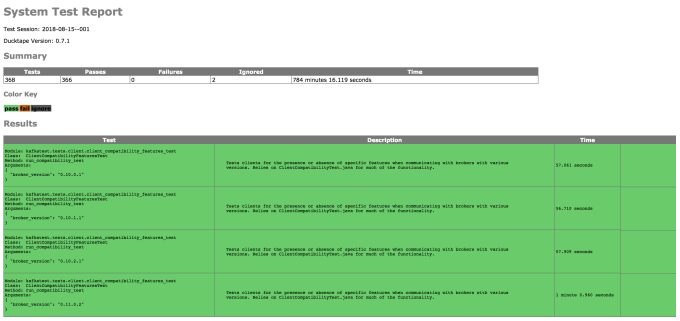
The authors also analyzed the determinism of failures. In distributed systems, there is the possibility that a sequence of events could produce a failure sometimes while not producing a failure other times. The authors found that 74% of failures are deterministic. This is a promising result because it means the failures are not by determined by the timing of the input events.

By far the most unexpected result of this paper was that 92% of errors in distributed systems are caused by incorrect handling of errors. Put another way, the system identified a fault could occur, but then the system handled it incorrectly. Further breaking down catastrophic failures, the authors found that 35% of catastrophic failures were caused by trivial mistakes, e.g., a "TODO" in the error handler or an ignored error.

To rectify these errors, the authors built a static checker, Aspirator, which checks a codebase for trivially incorrect error handling. We were not able to install Aspirator ourselves because the binaries are no longer hosted, but we did replicate some of Aspirator's functionality with simple searches. We found that in Kafka, there is at least one error handler with a "TODO" which would have caused Aspirator to fail.

1.2 Confluent Testing Effort for Kafka

A major vendor of Kafka, Confluent, has invested heavily in testing Kafka which they have documented [1]. The Kafka testing process begins when any change is proposed. When a feature is proposed, the featureś design is described by a design document called a Kafka Improvement Proposal KIP.


System Test Report

Test Session: 2018-08-15--001

Ducktape Version: 0.7.1

Summary

Tests	Passes	Failures	Ignored	Time
368	366	0	2	784 minutes 16.119 seconds

Color Key

pass fail ignore

Results

Test	Description	Time	

Module: kafkatest.tests.client.client_compatibility_features_test
Class: ClientCompatibilityFeaturesTest
Method: run_compatibility_test
Arguments:
{
"broker_version": "0.10.0.1"
}
	Tests clients for the presence or absence of specific features when communicating with brokers with various versions. Below we ClientCompatibilityTest.java for each of the functionality.	57.861 seconds	

Module: kafkatest.tests.client.client_compatibility_features_test
Class: ClientCompatibilityFeaturesTest
Method: run_compatibility_test
Arguments:
{
"broker_version": "0.10.1.1"
}
	Tests clients for the presence or absence of specific features when communicating with brokers with various versions. Below we ClientCompatibilityTest.java for each of the functionality.	58.710 seconds	

Module: kafkatest.tests.client.client_compatibility_features_test
Class: ClientCompatibilityFeaturesTest
Method: run_compatibility_test
Arguments:
{
"broker_version": "0.10.2.1"
}
	Tests clients for the presence or absence of specific features when communicating with brokers with various versions. Below we ClientCompatibilityTest.java for each of the functionality.	57.939 seconds	

Module: kafkatest.tests.client.client_compatibility_features_test
Class: ClientCompatibilityFeaturesTest
Method: run_compatibility_test
Arguments:
{
"broker_version": "0.11.0.2"
}
	Tests clients for the presence or absence of specific features when communicating with brokers with various versions. Below we ClientCompatibilityTest.java for each of the functionality.	1 minute 0.960 seconds	

Figure 1: Sample of Kafka test report

Once the KIP is accepted, the feature is then built. The built feature then goes through a thorough code review process where core developers examine the patch line-by-line.

In terms of actual testing, the Kafka project uses three kinds of tests: unit tests, integration tests and distributed systems tests. Kafka has at least 6,800 unit tests and 600 integration tests in the code base. There is no mention of their code coverage.

The third type of test mentioned by Confluent, distributed systems tests, are relatively novel. These types of tests involve putting the whole system under load and then injecting faults. Kafka has 310 of these type of tests which they run nightly. They then test the results of these tests online nightly. A sample from a recent test report is shown in **??**. These distributed tests are the reason that Confluent built Duck Tape which is discussed in detail above.

2 Evaluation

We showed that testing distributed systems can be challenging and the tooling for in the field is not great. Our first approach, using Duck Tape, did not work at all for two reasons.

First, Duck Tape does not work for our preferred version of python. This is indicative of a project which is not well-maintained.

Second, the Duck Tape framework only worked for positive test cases. If there was a test case which failed, Duck Tape would crash.

In terms of our own custom testing framework, we found that our framework worked well for simple test cases. We were able to successfully test that messages were posted to Kafka brokers. Further, we were able to produce a node failure in the cluster mid-test which proved Kafka's failover ability. Proving that these simple, but essential functions worked was an honest success for this project.

In terms of areas for improvement, there are a three things we would like to explore.

First, we would like to test the system under load. This would involve saturating the connection to the brokers with messages. Once the system is under load, we would like to inject faults by stopping nodes.

Second, we would like to simulate network failures. In our tests, we were only able to inject faults via node failure. If we had control over the network that the cluster was running on, we could

randomly drop packets and observe how the system reacted.

Third, we would like to automate this testing infrastructure by creating a fresh cluster for each test case. Configuration and state could be inherited from test-to-test, so by creating a new cluster for each test, via Jenkins and Kubernetes, we could isolate failures for proper debugging.

3 Conclusion

In all, we have explored the area of testing in distributed systems. By investigating the testing options available to one of the largest open-source distributed systems project, Kafka, we were able to find that the field of testing in distributed systems is relatively unexplored.

We started with the primary testing framework available for Kafka, Duck Tape. After attempting to use the framework, we found that it is not ready to ready for widespread use.

We developed our own testing framework which performed well for a limited set of simple test cases. However our testing framework is not well-suited for complex tests which would present as faults in a distibuted system.

4 Duck Tape

5 Custom Testing Apparatus

References

[1] Inc. Confluent. How apache kafka is tested. https://www.confluent.io/blog/apache-kafka-tested/.

[2] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, volume 11, pages 249–264. USENIX, 2014.

A Github Code

https://github.com/howinator/software-testing-kafka-project