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

During the semester, we have explored different aspects of software testing on a monolithic software application but how does this apply to the highly distributed systems of today's computing platforms. We aimed to explore this broad field by focusing on how to take lessons learned in class and apply them to our professional workplace in distributed systems.

As the authors, we have worked in distributed systems with both of our employers using cloud computing platforms and furthermore, distributed systems in production environments. However, neither one of our employers are utilizing a testing framework to confirm proper behavior of these systems. Why?

Testing in distributed systems is hard and requires specialized skills and resources to do it in repeatable, and reliable fashion. For example, testing in a monolithic software application has a very general testing pattern of instantiate class/method/object, pass in parameters and validate. However, this pattern doesn't work in distributed systems because a key principle that defines what a distributed system is redundancy and failover. In order to achieve redundancy and failover, a distributed application or system must be deployed in a cluster. This means when we pass in a value, any node in the cluster could act on the value which means we have to validate the cluster as a whole since we have no insights on which node or application is executing on that value.

This a vast problem but for the purpose of this paper we are going to prove out some common testing principles by building a simple testing framework that has the potential to be expanded to included more complex test cases.

1 Introduction

Before we embarked on the development of our very own test cases and testing framework, we wanted to explore existing approaches that companies are utilizing to test their cloud platforms or software systems. The research was promising but eye opening as most of these companies, like Netflix and Confluent, have millions of dollars and years to build these tools. However, we have neither, so we took our time to evaluate some tools such as Chaos Monkey by Netflix, Trogdor by the Kafka Community, and Ducktape by Confluent. These tools were designed by the parent company or community to solve some very complex problems in distributed systems like node failures, faulty internal communications and mocking production environments. These are common failures for distributed systems but we are targeting a simplified testing framework that could be modified for other potential failures or use cases. For example, after the infrastructure and environment is setup how do we simply test that a distributed application is setup correctly and accepting inputs like Unit Tests for monolithic applications.

During our research, we wanted to build on

what has been done in the industry but modify it to accomplish our simplified testing framework. This is easier said than done and we encounter a few failures before getting it right.

2 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. [3].

2.1 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.

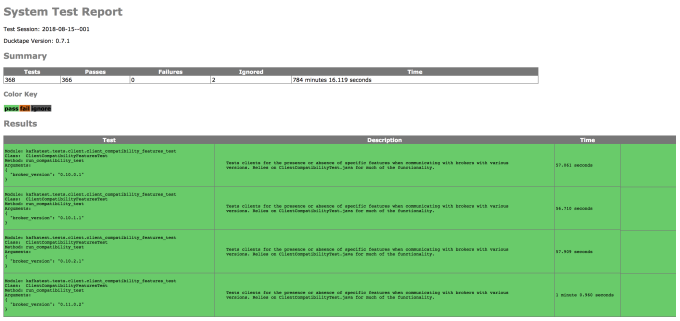
The authors also analyzed the determinism of failures. In distributed systems, there is the possibility that a sequence of events could produce a failure sometimes while not producing a failure other times. The authors found that 74% of failures are deterministic. This is a promising result because it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

2.2 Confluent Testing Effort for Kafka

A major vendor of Kafka, Confluent, has invested heavily in testing Kafka which they have documented [2]. 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. Refer to 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. Refer to 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. Refer to 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. Refer to 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 the figure above. These distributed tests are the reason that Confluent built Ducktape which is discussed above.

3 Kafka

Before we dive into the implementation details of our approach, we would like to explain what technologies we used and how they interact within our testing framework. At the infrastructure layer, we are utilizing Amazon Web Services, a cloud service provider that offers pay-as-you-go computing instances in technical terms called EC2 or Elastic Compute Service. These EC2 instances are what we have installed our Kafka environment on. Utilizing several instances of AWS's base service virtual machines, EC2 or Elastic Compute Service, as the environment for Kafka.

The Kafka environment is made up of several distributed applications as well as our own custom built software applications (producer and consumer) to perform the testing. [1] The first distributed application, Apache Kafka, is a horizontally scalable, distributed message broker. This is the same application used in numerous fortune 500 companies to pipe real-time data to streaming applications. See the figure below for a visual representation of the environment. In our tests we have setup a cluster of 3 Apache Kafka nodes.

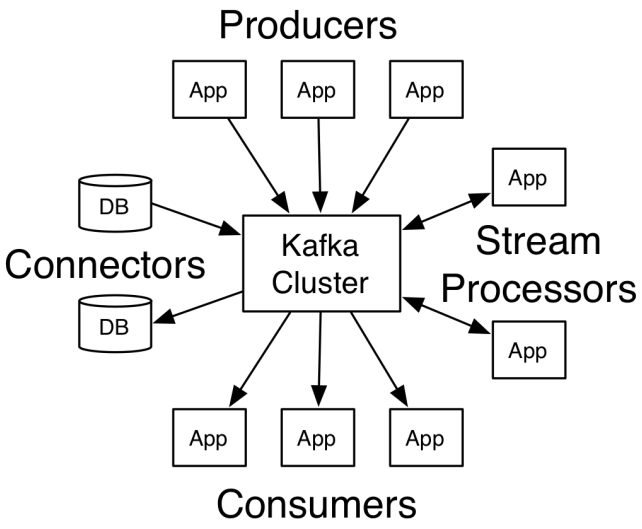


Figure 2: Kafka's Architecture [1]

Following the setup the Apache Kafka cluster, we need the ability to programmatically manage and communicate to the Kafka nodes. This led us to the most commonly used centralized service called Apache ZooKeeper. An important installation note for ZooKeeper is the application must be installed in an odd number of nodes because if a ZooKeeper cluster has a deadlock. The nodes will vote within the cluster to break the tie.

After the two core components were installed and configured, we built two python applications that utilized Kafka Producer and Consumer libraries to interact with the Kafka cluster and execute our test cases. The Kafka Producer is a software application that feeds the Kafka Cluster, also known as the Broker, the messages for storage. The producer can read from a variety of sources, our application reads the test cases from a file, and process them for sending to the Broker.

The Consumer is similar to the Producer as it is a custom-built software application but it performs the opposite data flow from the Producer. Once the message is stored on the Broker, a Kafka Consumer is deployed to pull the messages and process them. In our framework, we have built the Consumer to pull from the Broker and validate against the test cases hosted in the same file the Producer used.

4 Leveraging Confluent's Ducktape Framework

Confluent, a company provising Kafka as a Service, built a testing framework for Kafka called Ducktape. During our evaluation of the product, we severely struggled to get the framework to run. Mostly due to the lack of public documentation and resources for the tool, so we tried to reverse engineer the tool through source code hosted in the Github repository. After several days, we did get the library to execute in Python 2.7 but, only to discover, that out of the 33 tests embedded in Github repository, 23 worked properly.

The problem discovered with the remaining 10 tests were how they handle test failures. For example, in the file `test_failing_testspy`, we found that it raises an error when it detects the test fails. This is great except for the fact that when the user executes the entire test suite, the test suite will stop execution after test 23. Due to this step back, we decided to build our own testing framework and reduce the number of dependencies on external libraries.

5 Custom Testing Architecture

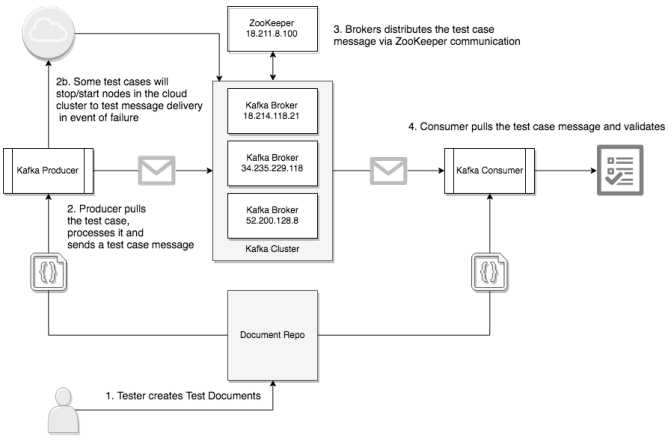


Figure 3: Diagram of Testing Architecture

The illustration above calls out how our testing framework executes see below for the execution path of the two test cases below.

5.1 Test Case: Validate values in a Stable Kafka Cluster

A Tester creates a text file and inserts the test criteria on a single line. For this test case, we used Instructions: Stable; Message: "Test Message" The Tester stores the test file, with the .txt extension, in the local file system. Both the Producer and Consumer must have permissions to access the file. The Tester executes both the Producer and the Consumer, written in Python. During the execution of the Producer application, the process opens the test file and sends the message ("Test Message") to a random node in the Kafka cluster. Once the Broker receives the message, it will duplicate the message across the cluster for redundancy. After the duplication occurs, the Consumer application has the ability to pull the message ("Test Message") and validate the message in the Test Case file. If the validation succeeds, the console prints out "Test Succeed, received: Test Message and expected Test Message"

5.2 Test Case: Validate values with a Kafka Node Failure

A Tester creates a test file and inserts the test criteria on a single line. For this test case, we used Instructions: Kill Node 2; Message: "Test Message 2" The Tester stores the test file, with the .txt extension, in the local file system. Both the Producer and Consumer must have permissions to access the file. The Tester executes both the Producer and the Consumer, written in Python. During the execution of the Producer application, the process opens the test file and sees the Kill Node 2 instructions which kicks off an API call to Amazon Web Services to kill the virtual machine that has a name Node 2. The Producers proceeds to send the message ("Test Message 2") to a different node in the Kafka cluster. Once the Broker receives the message, it will duplicate the message across the cluster for redundancy. After the duplication occurs, the Consumer application has the ability to pull the message ("Test Message 2") and validate the message in the Test Case file. If the validation succeeds, the console prints out "Test Succeed, received: Test Message 2 and expected Test Message 2"

Troubleshooting Help: Kafka Configuration: Each Kafka node has to be manually configured to have a unique node id, unique hostname with open port for Producer/Consumer communication, and

host address with open port for the ZooKeeper node. Running Kafka as Daemon: The ZooKeeper application automatically executes as a daemon across the cluster but the Kafka application must be executed with the "-daemon" parameter to execute correctly. Java Heap Space: Each Kafka node in a Broker cluster utilizes in-memory caching for quick retrieval of messages stored. This memory can quickly fill up with log files, process queues and messages, causing an unnecessary Java Heap Space Exception. To solve this, we adjusted our virtual machine memory from 2 GB to 8GB of RAM and storage from 10 GB SSD to 30 GB SSD.

6 Evaluation

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

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

Second, the Ducktape framework only worked for positive test cases. If there was a test case which failed, Ducktape 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.

7 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, Ducktape. 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.

References

[1] Inc. Apache. Introduction to kafka. https://kafka.apache.org/intro.html.

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

[3] 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