Title: Software Testing in Distributed Systems: Apache Kafka

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, either 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 define 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 is a huge problem but for the purpose of this paper we are going to prove out some simple testing principles by building a few simple tests to provide example of how this can be achieved. The tests we are writing are to validate proper behavior for a very common distributed messaging broker called Apache Kafka.

Keywords:

Distributed Systems, Software Testing, Apache Kafka

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 tools are great and I highly recommend you check out how they work but we wanted something even simpler. 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.

Since nothing that simple existed, we wanted to build something to simply do just that. However, we still wanted to borrow what has been done by the industry so we took a few different approaches before landing on our current approach.

Failed Approach to Leverage Confluent’s Ducktape framework

Confluent is a company that provides Kafka as Service meaning, companies pay Confluent to provide a high level of support for Apache Kafka. This level of support requires a great deal of insurance that Confluent’s managed Kafka Service is tested and runs accurately. Out of this requirement, Confluent built Ducktape (REF).

Ducktape is a testing framework for you to simulate or connect to a x node Kafka cluster and execute a series integration, failover and performance tests. Confluent advertises close to 6,800 unit tests and over 600 integration tests for their Kafka service (Ref).

However, 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\_tests.py, 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. This led us to the next approach.

Due to this failure, we decided to build our own testing framework and execute a few simple test cases.

Background

Before we dive into the implementation details of our approach, I would like to explain what technologies we used and they interact within our testing framework. At the infrastructure layer, we are utilizing Amazon Web Services, which is a cloud service provider that offers pay-as-you-go computing power. The computing power we are using is there base service virtual machines called EC2 or Elastic Compute Service. These EC2 instances are what we have installed our Kafka environment on.

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

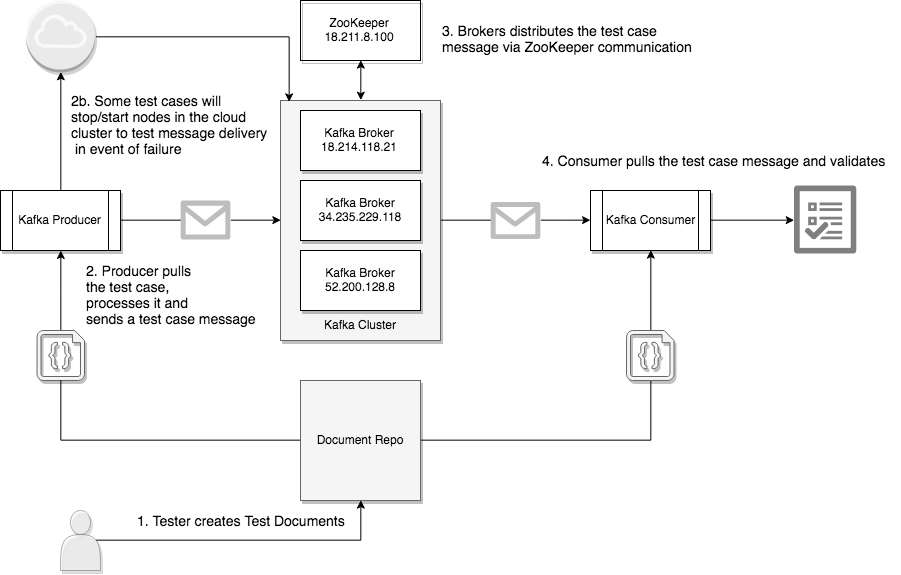


Following the Apache Kafka nodes, we have to be able to manage and communicate to the Apache Kafka nodes so we setup a single node of Apache ZooKeeper which is acting as the centralized service for maintaining information of the Kafka Nodes. An important note for those looking into ZooKeeper for their own projects, the ZooKeeper application must be installed in an odd number of nodes. For example, 1, 3,5 because if a ZooKeeper cluster has a deadlock based on voting then the cluster must be able to break the tie.

After the two core components are installed and configuration, 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 Broker, the messages for storage. The producer can read from a variety of sources and process them for sending to Broker. In our framework, we have built this to read the test cases from a file and send them to the Broker.

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

Illustration



Troubleshooting Issues:

Running Kafka as Daemon

Java Heap Space

Evaluation - Howie

How well did this work?

Improvements?

Related work - Howie

Confluent

Research Papers?

Conclusion - Howie

References

<https://kafka.apache.org/intro.html>

<https://zookeeper.apache.org/>

<https://www.confluent.io/>

<https://ducktape-docs.readthedocs.io/en/latest/#>

<https://www.confluent.io/blog/apache-kafka-tested/>