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, 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 few simple tests to provide framework of how this can be achieved.

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 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.

Approach to Leverage Confluent’s Ducktape framework

Confluent is a company that provides Kafka as a Service. In other words, 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 an n 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. Due to this step back, we decided to build our own testing framework and reduce the number of dependencies on external libraries.

Background

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. 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.

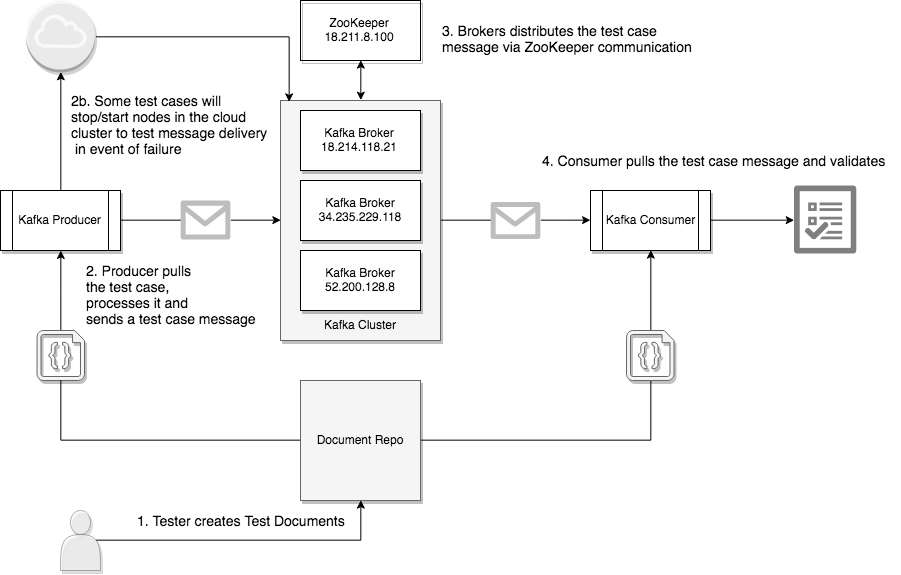


Following the setup the Apache Kafka cluster, we need the ability to programmatically manage and communicate to the Kafka nodes. This led us to the most commonly used centralized service called Apache ZooKeeper. An important installation note for ZooKeeper is the application must be installed in an odd 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.

Illustration



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

Test Case: Validate values in a Stable Kafka Cluster

1. 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”
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.
3. The Tester executes both the Producer and the Consumer, written in Python.
4. 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.
5. Once the Broker receives the message, it will duplicate the message across the cluster for redundancy.
6. 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.
7. If the validation succeeds, the console prints out “Test Succeed, received: Test Message and expected Test Message”

Test Case: Validate values with a Kafka Node Failure

1. 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”
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.
3. The Tester executes both the Producer and the Consumer, written in Python.
4. 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.
5. The Producers proceeds to send the message (“Test Message 2”) to a different node in the Kafka cluster.
6. Once the Broker receives the message, it will duplicate the message across the cluster for redundancy.
7. 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.
8. If the validation succeeds, the console prints out “Test Succeed, received: Test Message 2 and expected Test Message 2”

Troubleshooting Help:

1. 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.
2. 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.
3. 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.

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/>