CONNECTING TO APACHE KAFKA

Design and Deployment Considerations for Deploying Apache Kafka on AWS

Author:Alex Loddengaard
July 28, 2016

Apache Kafka (the basis for the Confluent Platform) delivers an advanced platform for streaming data used by hundreds of companies, large and small. Amazon Web Services (AWS) provides a powerful set of infrastructure services for a wide range of applications. Utilizing Kafka on AWS resources in a manner that delivers stability and performance of your Kafka cluster and the related Confluent components can be difficult. In this blog, we create an outline our recommendations for deploying, monitoring, and managing your Kafka cluster in AWS. In subsequent blogs we will detail how you can leverage Kafka as a bridge for cloud migrations.

1: Choose EBS or instance storage based on your application and operational needs

First, some Kafka background. Kafka's high-level abstraction is a topic — similar to a table in a database. A topic has one or many partitions for parallelism and scalability. Each partition is replicated to multiple brokers for redundancy. Kafka brokers store these topic partition replicas locally on disk. Sometimes a topic partition replica is referred to as a "log." To learn more, read about Kafka topics and partitions, and replication.

The *log.dirs* configuration specifies where the broker stores its logs (topic partition replicas). In AWS, both instance storage and EBS (Elastic Block Store) work well. However, each has its tradeoffs that may or may not be ideal for your application and operational needs. Below we'll go through each option and the associated tradeoffs.

EBS

In summary, using EBS volumes will decrease network traffic when a broker fails or is replaced. Also, the replacement broker will join the cluster more quickly. However, EBS volumes add cost to your AWS deployment.

Kafka has built-in fault tolerance by replicating partitions across a configurable number of brokers. However, when a broker fails and a new replacement broker is added, the replacement broker fetches all data the original broker previously stored from other brokers in the cluster that host the other replicas. Depending on your application, this could involve copying tens of gigabytes or terabytes of data. Fetching this data takes time and increases network traffic, which could impact the performance of the Kafka cluster for the period the data transfer is happening.

EBS volumes are persisted when an instance fails or is terminated. When an EC2 instance running a Kafka broker fails or is terminated, the broker's on-disk partition replicas remain intact and can be mounted by a new EC2 instance. By using EBS, most of the replica data for the replacement broker will already be in the EBS volume and hence won't need to be transferred over the network. Only data produced since the original broker failed or was terminated will need to be fetched across the network.

Most Kafka on AWS deployments use a replication factor of three. However, EBS does its own replication under the covers for fault tolerance. As such, using a Kafka replication factor of two can save storage costs. However, keep in mind that although the data is stored in a fault tolerant way, if two Kafka brokers go down, the partitions they each store will be offline.

EBS can be a single point of failure within an availability zone. If the broad EBS service within an availability zone has an outage, all brokers in that availability zone will be affected. See point #5 and #6 to learn more about cross-availability zone deployment recommendations.

Instance storage

In summary, using instance storage is cheaper but recovering from a failed or terminated broker will take longer and require more network traffic. This behavior is explained above. In general, instance storage is recommended for larger Kafka clusters.

EBS vs instance storage performance

In a simple Kafka benchmark, we saw better performance with *st1* EBS than instance storage. *st1* EBS is optimized for throughput.

Replacing a Failed Broker

Regardless of what type of storage is used, when replacing a Kafka broker, the recommended practice is to use the broker ID from the failed broker in the replacement broker. Doing so is the most operationally simple approach, because the replacement broker will resume the work that the failed broker was doing automatically.

2: Isolate Kafka data to dedicated disks

Kafka brokers should be configured with dedicated disks, to limit disk thrashing and increase throughput. More specifically, *log.dirs* should only contain disks (or EBS volumes) that are dedicated to deploy Kafka. Don't include the operating system's disk, or any other disks that are used for other purposes.

3: Choose the right instance type

We recommend using *d2.xlarge* if you're using instance storage, or *r3.xlarge* or *m4.2xlarge* if you're using EBS. These instance types are inline with our general Kafka hardware recommendations. This isn't specific to Kafka, but you might want to consider buying reserved instances to lower costs.

4: Configure the network correctly

AWS network configuration with Kafka is similar to other distributed systems, with a few caveats mentioned below. AWS offers a variety of different IP and DNS options. Choose an option that keeps inter-broker network traffic on the private subnet and allows clients to connect to the brokers. Note that inter-broker and client communication use the same network interface and port. We recommend reading the EC2 instance IP addressing documentation to learn more.

When a broker is started, it registers its hostname with ZooKeeper. The producer since Kafka 0.8.1 and the consumer since 0.9.0 are configured with a bootstrapped (or "starting") list of Kafka brokers. Prior versions were configured with ZooKeeper. In both cases, the client makes a request (either to a broker in the bootstrap list, or to ZooKeeper) to fetch all broker hostnames and begin interacting with the cluster.

Depending on how the operating system's hostname and network are configured, brokers on EC2 instances may register hostnames with ZooKeeper that aren't reachable by clients. The purpose of *advertised.listeners* is to address exactly this problem; the configured protocol, hostname, and port in *advertised.listeners* is registered with ZooKeeper instead of the operating system's hostname.

In a multi-region architecture, careful consideration has to be made for MirrorMaker. Under the covers, MirrorMaker is simply a consumer and a producer joined together. If MirrorMaker is configured to consume from an Elastic IP (EIP), the single broker tied to the EIP will be reachable, but the other brokers in the source cluster won't be. MirrorMaker needs access to all brokers in the source and destination region, which in most cases is best implemented with a VPN between regions. For more details please refer to Connecting Multiple VPCs with EC2 Instances in the AWS documentation.

Client service discovery can be implemented in a number of different ways. One option is to use HAProxy on each client machine, proxying *localhost* requests to an available broker. Synapse works well for this. Another option is to use Route 53 for DNS. In this setup, the TTL must be set in the client JVM to get around DNS caching. See setting the JVM TTL in the AWS documentation. Another option is to use an Elastic Load Balancer (ELB). In this configuration, ensure the ELB is not public to the internet. Sessions and stickiness do not need to be configured because Kafka clients only make a request to the load balancer at startup. A health check can be a ping or a telnet.

5: Distribute Kafka brokers across multiple availability zones

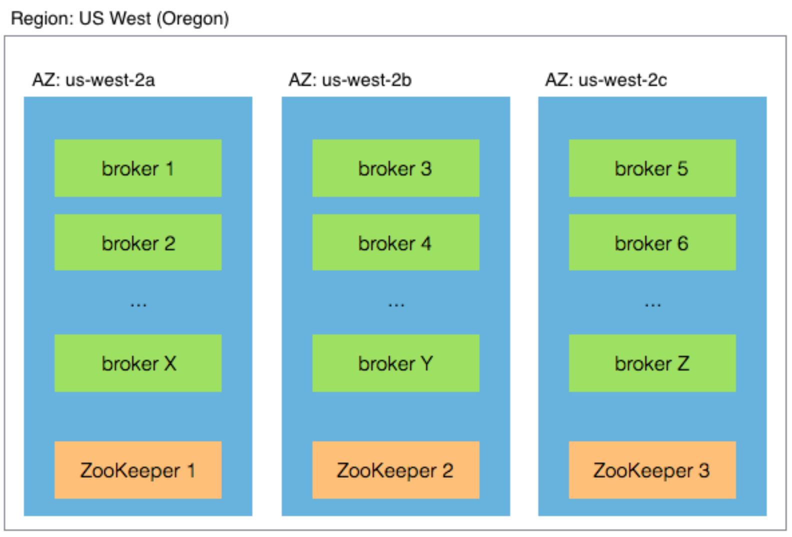
Kafka was designed to run within a single data center. As such, we discourage distributing brokers in a single cluster across multiple regions. However, we recommend "stretching" brokers in a single cluster across availability zones within the same region.

A multi-availability zone cluster offers stronger fault tolerance because a failed availability zone won't cause Kafka downtime.

However, in this configuration, prior to Kafka 0.10, you must assign partition replicas manually to ensure that replicas for each partition are spread across availability zones. Replicas can be assigned manually either when a topic is created, or by using the *kafka-reassign-partitions* command line tool. Kafka 0.10 supports rack awareness, which makes spreading replicas across availability zones much easier to configure.

Note that cross-availability zone data transfer fees will apply.

Below is what a multi-availability zone architecture looks like (more on ZooKeeper later):



6: Distribute ZooKeeper nodes across multiple availability zones

Similar to #5 above, ZooKeeper should be distributed across multiple availability zones as well, to increase fault tolerance. To tolerate an availability zone failure, ZooKeeper must be running in at least three different availability zones. In a configuration where three ZooKeepers are running in only *two* availability zones, if the availability zone with two ZooKeepers fails, ZooKeeper will not have quorum and will not be available.

See the architecture diagram above and this article on building a global, highly available service discovery infrastructure with ZooKeeper.

7: Monitor broker performance and terminate poorly performing brokers

Individual EC2 instance performance can decrease unexpectedly over time for unknown reasons.

We recommend terminating and replacing a broker if the 99 percentile of produce/fetch request latency is higher than is tolerable for your application.

For a great video on how Netflix deployed Kafka on AWS, watch the recording of “Kafka at Scale in the Cloud” from Kafka Summit presented by Allen Wang, Software Engineer, Netflix.