Achieving Fault Tolerance with Kafka: A Detailed Explanation

Debut Infotech Follow
Aug 20, 2018 · 7 min read

Fault tolerance is no alien to the blockchain. The technique is being used by a number of industry players, including IBM, Intel, Cisco, Microsoft, Fujitsu, NEC, and more. It's found in pretty much everything in different forms and generally refers to the property of a system (computer, network, or cloud cluster) to continue operating properly in

the event of an unexpected hardware or software failure. The failed components are automatically replaced with backup components, which are identical or equivalent to the original ones, ensuring no loss of data or productivity.

Enough of the general theory, now moving to the technical stuff!

What Exactly is Fault Tolerance and Why is it Essential?

In terms of blockchain, the concept of fault tolerance whirls around determining the number of hostile or uncooperative nodes a user can have before his or her network gets compromised. The limit varies from blockchain to blockchain. Bitcoin, for instance, is generally considered 51 percent fault tolerant (although some believe the percentage is quite a bit lower than that, i.e., between 25% and 33%). This basically means that the network will continue to be functional until at least 51 percent of Bitcoin's mining power is interrupted or broken.

The psychology behind creating a fault-tolerant system is to avert disruptions arising from a single point of failure or one single shared source, ensuring high availability and business continuity of critical corporate processes & functions. High availability refers to a system's ability to evade loss of efficiency by minimizing downtime. It's usually formulated in terms of a system's uptime, a percentage of its total running time to be precise. Five nines, or 99.999 percent uptime, is generally considered as the "*holy grail*" of availability.

Why Kafka for Fault Tolerance?

It's no secret that data is the main element of internet applications, which typically includes the following:

- Blog visits, page views, page visits and clicks;
- User activities;
- Events conforming to logins;
- Social media metrics and indicators such as likes, shares, retweets, comments, views, and so on;
- Application specific information such as logs, page load time, and performance.

In the present big-data era, one of the key challenges most organizations face is collecting copious amounts the data, which is a cumbersome and time-consuming process, and then analyzing it in real time, which often leads to data loss due to hardware failure or any fault in the system. An efficient way to solve this problem is by using messaging systems, which provide seamless integration between distributed applications with the help of messages.

One such messaging system that can be used for preventing data loss and ensuring fault tolerance is *Apache Kafka*.

For those who're unfamiliar with the term Kafka, it is a scalable, fault-tolerant, publish-subscribe messaging system that allows you to build distributed applications. It was initially developed at LinkedIn and later became a primary part of the Apache project. Kafka is lightning-fast, scalable, durable, and distributed by design. It is currently used by

numerous web-scale Internet enterprises such as LinkedIn, Twitter, Airbnb, Yahoo, Netflix, Uber, Spotify, Pinterest, Tumblr, Mozilla, Amazon Web Services, etc.

Big Data ingestion at Netflix:

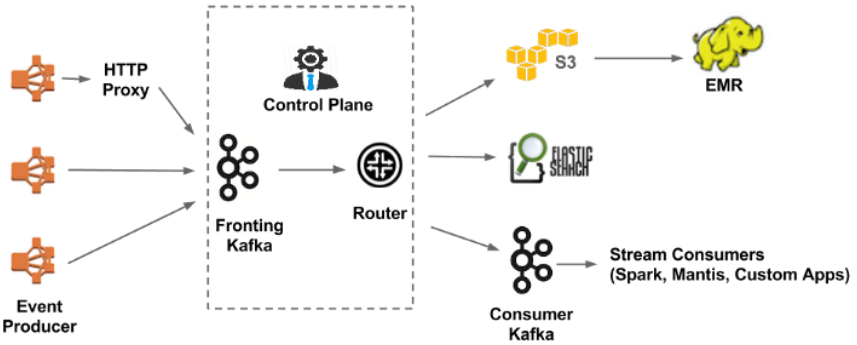


Image source: Netflix

The biggest upside of this powerful asynchronous messaging system is that it caters to both large-scale and typically slower-to-adopt, traditional enterprises. Whenever you write a message to Kafka, it is persisted and replicated to peer brokers for fault tolerance, and it stays around for quite a long time, i.e., for 7 days, 30 days, or even more.

How Can Fault Tolerance Be Achieved Through Kafka?

To understand this, let's first take a look at the Kafka architecture, which comprises Kafka Cluster, Zookeeper, Producers, and Consumers. The following illustration exhibits the architecture of Kafka.

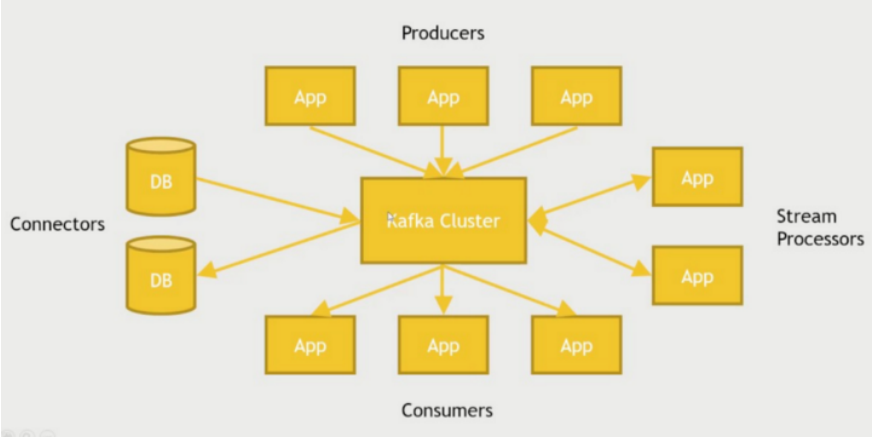


Image Source: YouTube

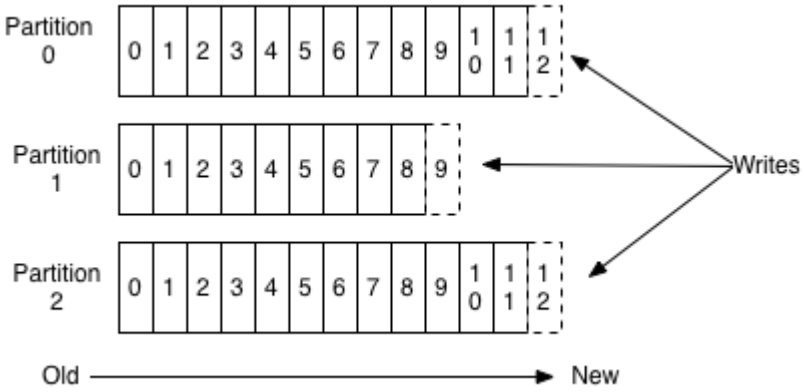
- A stream of messages of any particular category is referred to as a *"Topic"*.
- Producers are the clients or applications that publish messages to a Topic.
- Consumers are the applications that subscribe to topics and reads the messages.
- Kafka cluster consists of multiple servers, each of which is called a *"Broker"*.
- Brokers use Zookeepers, an open-source coordination service for distributed applications, for maintaining the cluster state, as they are stateless.
- Also, each Kafka broker coordinates with other Kafka brokers using a Zookeeper.

- Zookeepers notify both producers and consumers about the presence of a new broker or failure of the broker in Kafka system.

With Kafka, you can create multiple types of clusters such as Single Node Single Broker cluster, Single Node Multiple Broker cluster, and Multiple Nodes Multiple Broker cluster.

Now coming back to the original question, how Kafka helps you achieve fault tolerance? Here's how it does that!

For each topic, the Kafka cluster maintains a partitioned log, which looks like the image below:



Each partition is an ordered, immutable arrangement of messages that is recurrently added to a commit log. The partitions of the log are dispersed over the servers in the Kafka cluster with each server handling data and requests for a share of the partitions. Thus, Kafka

achieves fault tolerance by duplicating each partition over a number of servers.

Below is an example depicting how Kafka is configured and promotes fault tolerance.

Generally, there are two files within the Kafka config folder: zookeeper.properties and server.properties. Below we have portrayed the components of these two files using an example.

Let's cover the server.properties file first.

Server Basics:

```
# The id of the broker. This must be set to a unique integer
for each broker.

broker.id=1
```

Log Basics:

```
# A comma separated list of directories under which to store
log files

log.dirs=/home/web/kafka2.11/zookeeper
```

```
# The default number of log partitions per topic. More
partitions allow greater

# parallelism for consumption, but this will also result in
more files across

# the brokers.

num.partitions=3

# The number of threads per data directory to be used for
log recovery at startup and flushing at shutdown.

# This value is recommended to be increased for
installations with data dirs located in RAID array.

num.recovery.threads.per.data.dir=1
```

Internal Topic Settings:

```
# The replication factor for the group metadata internal
topics “__consumer_offsets” and “__transaction_state”

# For anything other than development testing, a value
greater than 1 is recommended for to ensure availability
such as 3.

offsets.topic.replication.factor=1

transaction.state.log.replication.factor=1
```

```
transaction.state.log.min.isr=1
```

Log Retention Policy:

```
# The minimum age of a log file to be eligible for deletion
due to age

log.retention.ms=-1

# The maximum size of a log segment file. When this size is
reached a new log segment will be created.

log.segment.bytes=1073741824

# The interval at which log segments are checked to see if
they can be deleted according

# to the retention policies

log.retention.check.interval.ms=300000
```

Zookeeper:

```
# Zookeeper connection string (see zookeeper docs for
details).
```

```
# This is a comma separated host:port pairs, each
corresponding to a zk

# server. e.g.
“127.0.0.1:3000,127.0.0.1:3001,127.0.0.1:3002”.

# You can also append an optional chroot string to the urls
to specify the

# root directory for all kafka znodes.

zookeeper.connect=zookeeper0:2181,zookeeper1:2181,zookeeper2
:2181

# Timeout in ms for connecting to zookeeper

zookeeper.connection.timeout.ms=6000
```

Group Coordinator Settings:

```
# The following configuration specifies the time, in
milliseconds, that the GroupCoordinator will delay the
initial consumer rebalance.

# The rebalance will be further delayed by the value of
group.initial.rebalance.delay.ms as new members join the
group, up to a maximum of max.poll.interval.ms.

# The default value for this is 3 seconds.

# We override this to 0 here as it makes for a better out-
of-the-box experience for development and testing.
```

```
# However, in production environments the default value of 3
seconds is more suitable as this will help to avoid
unnecessary, and potentially expensive, rebalances during
application startup.

group.initial.rebalance.delay.ms=0

unclean.leader.election.enable = false

min.insync.replicas=2

default.replication.factor=3
```

Now let’s cover the components of the zookeeper.properties file using an example.

```
# Licensed to the Apache Software Foundation (ASF) under one
or more

# contributor license agreements. See the NOTICE file
distributed with

# this work for additional information regarding copyright
ownership.

# The ASF licenses this file to You under the Apache
License, Version 2.0

# (the “License”); you may not use this file except in
compliance with

# the License. You may obtain a copy of the License at

#
```

```
# http://www.apache.org/licenses/LICENSE-2.0

#

# Unless required by applicable law or agreed to in writing,
software

# distributed under the License is distributed on an “AS IS”
BASIS,

# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
express or implied.

# See the License for the specific language governing
permissions and

# limitations under the License.

# the directory where the snapshot is stored.

dataDir=/home/android/kafka2.11/zookeeper

# the port at which the clients will connect

clientPort=2181

# disable the per-ip limit on the number of connections
since this is a non-production config

maxClientCnxns=0

tickTime=5000
```

```
initLimit=5

syncLimit=2

server.1=0.0.0.0:2888:3888

server.3=avinash:2888:3888

server.1=sanjay:2888:3888
```

It takes time to understand what's inside these files and what exactly it does. But once you'll figure that out, life becomes a breeze and you'll notice the difference it makes to your data. I'd personally recommend spending some time playing around different configs to get an idea about it.

That's all for now folks. To conclude, Kafka provides you with high availability, fault tolerance, and business continuity for different scenarios, which is why it's a must-have for all enterprises with data critical systems.

If you're looking for someone who could help you achieve fault tolerance with Kafka, then get in contact with us. We're a leading **blockchain development company in India** with a foothold in the US and Canada, offering a complete suite of web & mobile application fault tolerant solutions. We also offer a 99.999 percent uptime guarantee which reflects our confidence in the resiliency of our solution and the quality of our services.

Happy reading!