Confluent Extra Operations Exercises

Table of Contents

Introduction

These extra, optional Hands-On Exercises use the Operations Virtual Machine. We have installed Confluent Platform on the VM via RPM. You will automatically be logged in to the machine when it starts up.

Login

If you need the login credentials, they are:

```
Username: training
Password: training
```

The `training` user has passwordless `sudo` enabled.

If you wish, you can also ssh in to the VM from a terminal on your local machine by typing

```
ssh training@localhost -p 2222
```

However, you will probably find it easiest to do all your work within the VM's own graphical user interface.

You will find the files you need at the following locations:
* `/usr/bin/` – Driver scripts for starting/stopping services
* `/vagrant/configfiles/` – configuration files specific to this VM
* `/usr/share/java/kafka` – jars
* `/var/log/kafka` – Apache ZooKeeper logs
* `/var/log/kafka[0-n]` – Broker logs
* `/var/lib/kafka_[0-n]` – data files

The VM will run three broker instances to demonstrate the distributed nature of of Apache Kafka.

The VM also includes Graphite to monitor the JMX metrics for the Brokers. The statistics can be viewed on a Web browser from within the VM at `http://localhost:8000`, or on a Web browser on the host operating system at `http://localhost:4567`.

If you are viewing this PDF using Apple's Preview application on a Mac, you may find that copying and pasting commands into the VM does not work properly. This is a known issue with Preview; either copy each line separately, or use an alternative viewer such as Acrobat Reader.

Hands-On Exercise: Getting Set Up, Basic Operation, and Metrics

Managing Services

1. Launch the VM by double-clicking the .vbox file in the VM directory. When VirtualBox Manager appears, click on the name of the VM and select 'Start'.
2. Launch a terminal in the VM and run the training VM script to activate the required services in the VM.

```
$ bash /vagrant/configfiles/start-confluent.sh
```

This script will start the Confluent Platform services as well as Graphite.

3. Verify that three Brokers (`server0`, `server1`, `server2`) are running:

```
$ ps -Af | grep java | grep -o server[0-2]
server0
server1
server2
```

You may need to periodically truncate the data files. To do so,

Stop the Kafka service:

```
$ sudo /usr/bin/kafka-server-stop
```

Go into each of the data directories (`/var/lib/kafka_[0-n]`) and remove the log files:

```
$ sudo rm -rf /var/lib/kafka_*
```

Restart the services.

```
$ bash /vagrant/configfiles/start-confluent.sh
```

If you wish to stop or start individual Brokers, use the following commands:

To stop individual Brokers:

```
$ sudo ps -Af |grep java |grep server0 | tr -s ' ' \
|cut -f2 -d' ' | sudo xargs kill -15
```

```
$ sudo ps -Af |grep java |grep server1 | tr -s ' ' \
|cut -f2 -d' ' | sudo xargs kill -15
```

```
$ sudo ps -Af |grep java |grep server2 | tr -s ' ' \
|cut -f2 -d' ' | sudo xargs kill -15
```

To start individual Brokers:

```
$ sudo JMX_PORT=9990 LOG_DIR=/var/log/kafka0 kafka-server-start \
-daemon /vagrant/configfiles/server0.properties
```

```
$ sudo JMX_PORT=9991 LOG_DIR=/var/log/kafka1 kafka-server-start \
-daemon /vagrant/configfiles/server1.properties
```

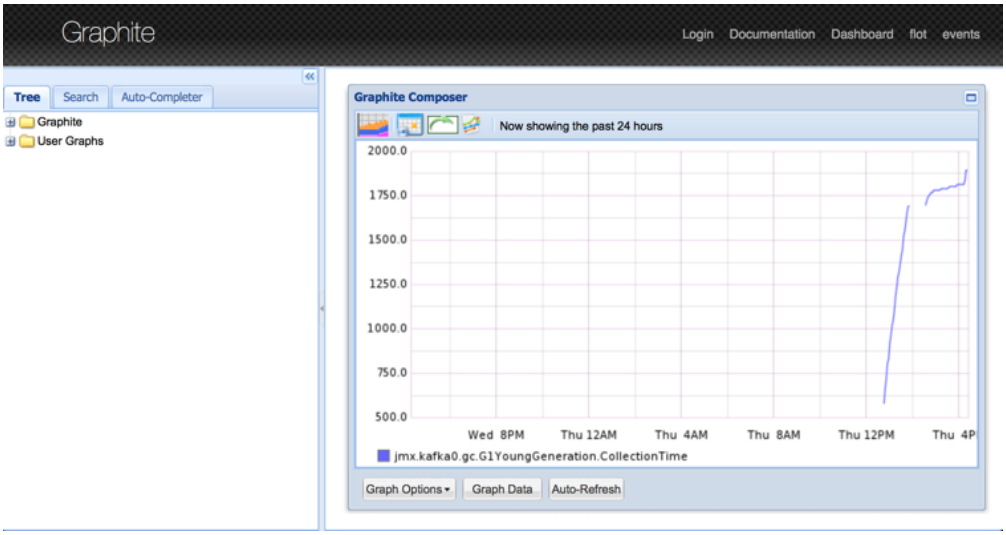
```
$ sudo JMX_PORT=9992 LOG_DIR=/var/log/kafka2 kafka-server-start \
-daemon /vagrant/configfiles/server2.properties
```

Working with Graphite

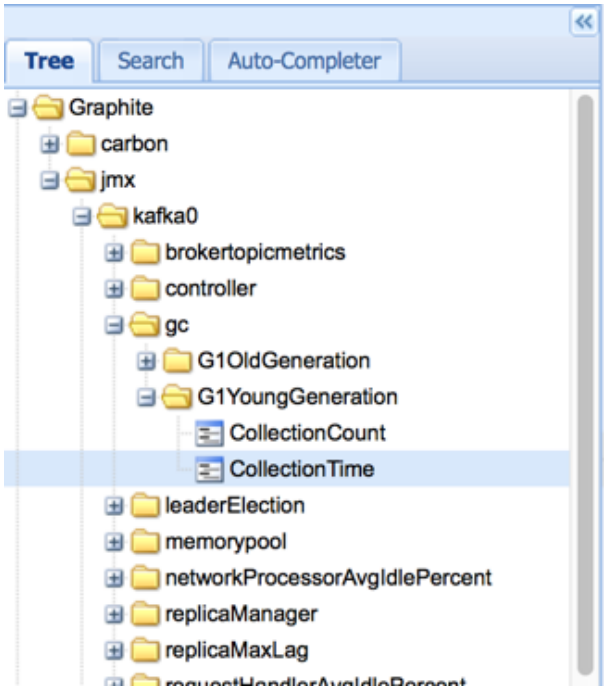
The Exercise environment is preinstalled with Graphite and related applications for performance monitoring.

4. Open a web browser in the VM (the circular Firefox icon at the top of the VM screen next to the System menu) and navigate to `http://localhost:8000`.

You can also use the Web browser on your host machine; in that case, the port number is 4567.

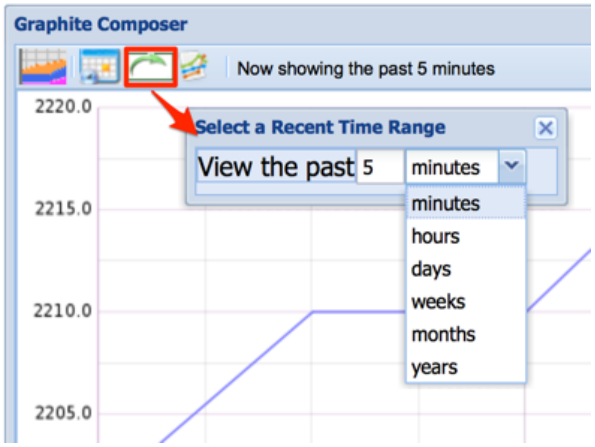


5. Open the navigation tree `Graphite/jmx/kafka0/gc/G1YoungGeneration/CollectionTime`

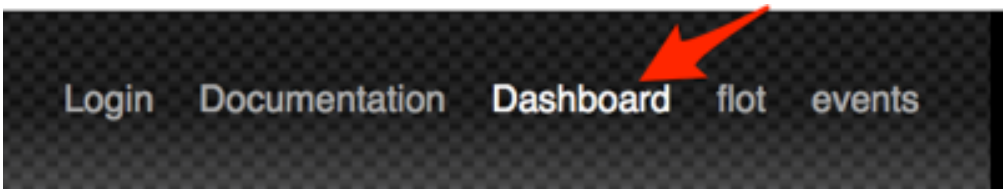


If the specified tree structure is not available, some of the services may still be starting up. Wait a few minutes, then refresh the browser and try again.

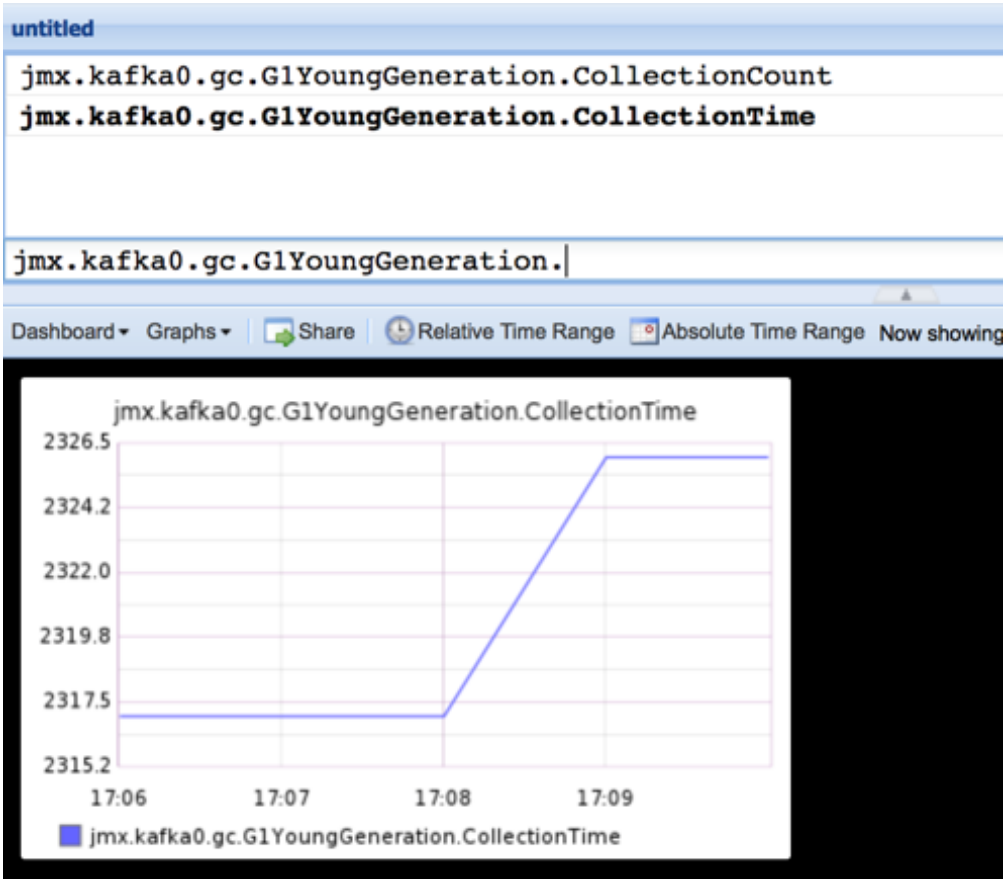
6. In the Graphite Composer, set the Recent Data time range to the last five minutes.



7. Click the Dashboard link in the top right hand corner of web page. This will take you to the Graphite dashboard viewer (`http://localhost:8000/dashboard/`).

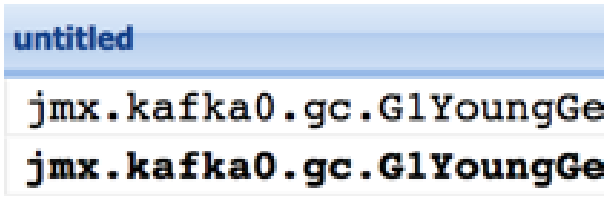


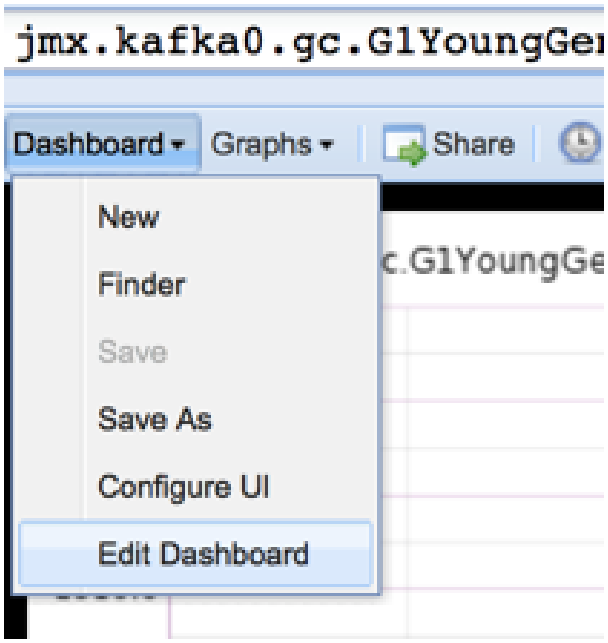
8. Use the links in the top half of select the same metric you were observing in the previous page.



9. Configure the Dashboard with multiple panes to track specific metrics.

a. In the lower half of the web page, select **Dashboard > Edit Dashboard**



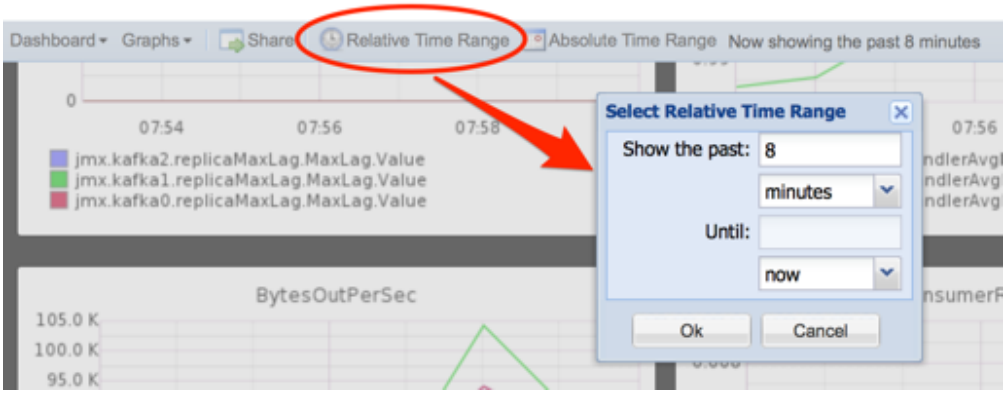


b. In a different browser window or tab, display the contents of `file:///home/training/kafka_dash.txt`

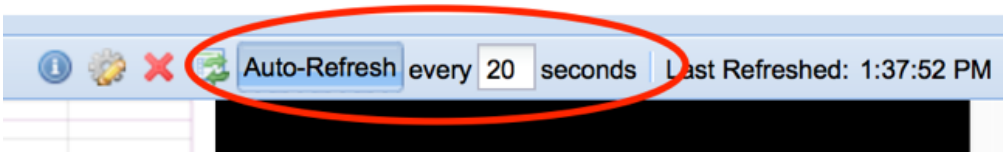
c. Copy and paste the contents of the `kafka_dash.txt` file to replace the Edit Dashboard dialogue. Click **Update (doesn't save)** to accept the changes.

d. Review the metrics tracked by the new panes in the dashboard. Some metrics will not populate until the cluster has been running for some time.

e. Click **Relative Time Range** to limit the view to the last 8 minutes.



f. Set Auto-Refresh rate to 20 seconds, then click the **Auto-Refresh** to activate.



Create a simple topic

10. Create a test topic with one partition and one replica.

```
$ kafka-topics \
--zookeeper localhost:2181 \
--create \
--topic i-love-logs \
--partitions 1 \
--replication-factor 1
Created topic "i-love-logs".
```

11. Verify that topic is created.

```
$ kafka-topics \
--zookeeper localhost:2181 \
--describe \
--topic i-love-logs
Topic:i-love-logs   PartitionCount:1    ReplicationFactor:1 Configs:
    Topic: i-love-logs  Partition: 0    Leader: 0   Replicas: 0 Isr: 0
```

Observe system load when running connectors

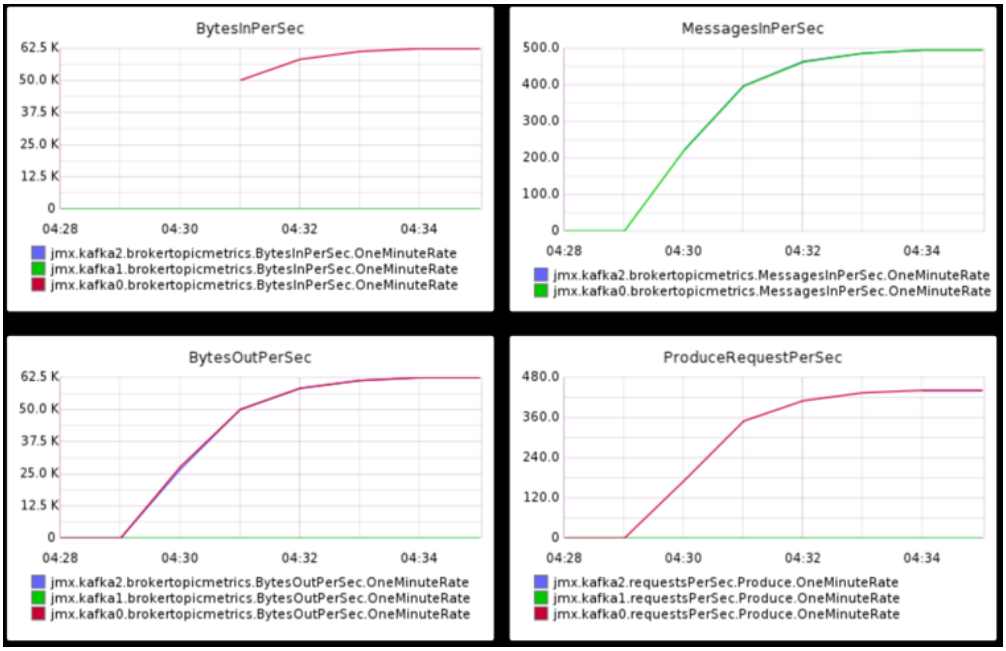
12. Produce some data to send to the topic `i-love-logs`. Leave this process running until instructed to end it.

The following command line sends 10 million messages, 100 bytes in size, at a rate of 1000 messages/sec.

```
$ kafka-producer-perf-test \
--topic i-love-logs \
--num-records 10000000 \
--record-size 100 \
--throughput 1000 \
--producer-props
bootstrap.servers=localhost:9090,localhost:9091,localhost:9092

4977 records sent, 994.8 records/sec (0.09 MB/sec), 31.9 ms avg latency,
397.0 max latency.
5030 records sent, 1005.6 records/sec (0.10 MB/sec), 7.7 ms avg latency,
107.0 max latency.
4998 records sent, 999.2 records/sec (0.10 MB/sec), 5.5 ms avg latency,
71.0 max latency.
5008 records sent, 1001.2 records/sec (0.10 MB/sec), 6.7 ms avg latency,
110.0 max latency.
5001 records sent, 999.8 records/sec (0.10 MB/sec), 6.3 ms avg latency,
97.0 max latency.
...
```

13. Observe the dashboard for five minutes. Pay attention to changes in `BytesInPerSec`, `MessagesInPerSec`, `BytesOutPerSec`, `ProduceRequestPerSec`.

After five minutes, the graphs should look similar to this:

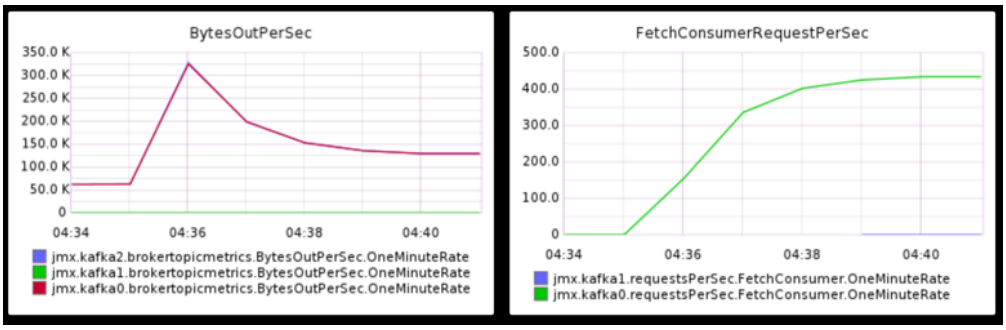


14. In another terminal, consume some data from the topic `i-love-logs`. Leave this process running until instructed to end it.

```
$ kafka-consumer-perf-test \
--broker-list localhost:9090 \
--topic i-love-logs \
--group test-group \
--messages 10000000 \
--threads 1 \
--show-detailed-stats \
--reporting-interval 5000 \
--new-consumer
time, data.consumed.in.MB, MB.sec, data.consumed.in.nMsg, nMsg.sec
2016-03-31 23:00:20:438, 0, 0.4768, 0.7882, 5000, 8264.4628
2016-03-31 23:00:20:797, 0, 0.9537, 1.3319, 10000, 13966.4804
2016-03-31 23:00:20:998, 0, 1.4305, 2.3842, 15000, 25000.0000
2016-03-31 23:00:21:121, 0, 1.9073, 3.9085, 20000, 40983.6066
2016-03-31 23:00:21:247, 0, 2.3842, 3.8147, 25000, 40000.0000
2016-03-31 23:00:21:297, 0, 2.8610, 9.7314, 30000, 102040.8163
2016-03-31 23:00:21:444, 0, 3.3379, 3.4553, 35000, 36231.8841
```

15. Observe the dashboard for 5 minutes. Pay attention to changes in `BytesOutPerSec`, `FetchConsumerRequestsPerSec`.

After five minutes, the graphs should look similar to this:



16. For both the consumer and the producer, go to the terminal window and press `Ctrl-c` to end each process.
17. Verify that the Kafka log4j logs have been created in the directories for each of the brokers (e.g., `/var/log/kafka0`, `/var/log/kafka1`, `/var/log/kafka2`).

```
$ ls /var/log/kafka0

controller.log         kafkaServer-gc.log  server.log
kafka-authorizer.log   kafkaServer.out     state-change.log
kafka-request.log      log-cleaner.log
```

STOP HERE. THIS IS THE END OF THE EXERCISE.

Hands-On Exercise: The Distributed Log, and Dynamic Recovery

In this Exercise, you will investigate the distributed log. You will then simulate the death of one of the Brokers, and see how to recover from Broker failure.

Observing the distributed log

1. Create a topic with six partitions and two replicas.

```
$ kafka-topics \
--zookeeper localhost:2181 \
--create \
--topic replicated-topic \
--partitions 6 \
--replication-factor 2
```

2. Describe the topic to see that it has been created correctly.

```
$ kafka-topics \
--zookeeper localhost:2181 \
--describe \
--topic replicated-topic
Topic:replicated-topic  PartitionCount:6    ReplicationFactor:2 Configs:
    Topic: replicated-topic Partition: 0    Leader: 0   Replicas: 0,1
Isr: 0,1
    Topic: replicated-topic Partition: 1    Leader: 1   Replicas: 1,2
Isr: 1,2
    Topic: replicated-topic Partition: 2    Leader: 2   Replicas: 2,0
Isr: 2,0
    Topic: replicated-topic Partition: 3    Leader: 0   Replicas: 0,2
Isr: 0,2
    Topic: replicated-topic Partition: 4    Leader: 1   Replicas: 1,0
Isr: 1,0
    Topic: replicated-topic Partition: 5    Leader: 2   Replicas: 2,1
Isr: 2,1
```

3. Start the console producer using the same topic we used earlier. Type “I heart Logs” and press Enter. Add a few more messages then press `Ctrl-C` to exit the producer.

```
$ kafka-console-producer \
--broker-list localhost:9090 \
--topic replicated-topic

I heart logs
Hello world
All your bases
<ctrl-c>

$
```

4. Navigate to one of the data file directories where Kafka keeps its logs. Examine the contents of two files, `recovery-point-offset-checkpoint` and `replication-offset-checkpoint`.

a. The `recovery-point-offset-checkpoint` file records the point up to which data has been flushed to disks. This is important as, on hard failure, we need to scan unflushed data, verify the CRC, and truncate the corrupted log.

```
$ cd /var/lib/kafka_0
$ grep "replicated-topic" recovery-point-offset-checkpoint
replicated-topic 2 0
replicated-topic 4 0
replicated-topic 0 0
replicated-topic 3 0
```

b. The `replication-offset-checkpoint` file is the offset of the last committed offset. On startup, the follower uses this to truncate any uncommitted data.

```
$ grep "replicated-topic" replication-offset-checkpoint
replicated-topic 2 0
replicated-topic 4 0
replicated-topic 0 1
replicated-topic 3 1
```

5. Examine the contents of one of the data directories.

```
$ cd replicated-topic-0
$ ls

00000000000000000000.index
00000000000000000000.log
```

You should see two data files. One is for the log itself. The other is for the index. The index maps the consumer offset to the physical offset within the log file.

6. Now use the DumpLogSegements tool to look at the metadata.

```
$ kafka-run-class kafka.tools.DumpLogSegments \
--print-data-log \
--files 00000000000000000000.log

Dumping 00000000000000000000.log
Starting offset: 0
offset: 0 position: 0 isvalid: true payloadsize: 100 magic: 0
compresscodec: NoC
ompressionCodec crc: 3514299894 payload:
offset: 1 position: 126 isvalid: true payloadsize: 100 magic: 0
compresscodec: N
oCompressionCodec crc: 3514299894 payload:
offset: 2 position: 252 isvalid: true payloadsize: 100 magic: 0
compresscodec: N
oCompressionCodec crc: 3514299894 payload:
offset: 3 position: 378 isvalid: true payloadsize: 100 magic: 0
compresscodec: N
...
offset: 191544 position: 50339520 isvalid: true payloadsize: 100 magic: 0
compresscodec: NoCompressionCodec crc: 3514299894 payload:
offset: 191545 position: 50339646 isvalid: true payloadsize: 100 magic: 0
compresscodec: NoCompressionCodec crc: 3514299894 payload:

$ sudo kafka-run-class kafka.tools.DumpLogSegments \
--files  00000000000000000000.index

Dumping 00000000000000000000.index
offset: 98 position: 12348
offset: 138 position: 17388
offset: 175 position: 22050
offset: 212 position: 26712
offset: 258 position: 32508
offset: 304 position: 38304
offset: 339 position: 42714
...
offset: 191469 position: 24125094
offset: 191502 position: 24129252
offset: 191535 position: 24133410
```

Questions:

.. Can you work out what the ‘offset' of the message “I heart Logs” that you entered earlier is?
.. What is the position of this offset in the file?

Clue: if you can’t find the value in the replicated-topic-0 directory, where else might it be?

Killing a Broker

7. Generate some load on the system:

```
$ kafka-producer-perf-test \
--topic replicated-topic \
--num-records 10000000 \
--record-size 100 \
--throughput 1000 \
--producer-props
bootstrap.servers=localhost:9090,localhost:9091,localhost:9092
```

8. In a different terminal window, stop Broker 0. Verify that there are only two running servers before continuing with the instructions.

```
$ ps -Af | grep java |grep server0 | tr -s ' ' \
|cut -f2 -d' ' | sudo xargs kill -15

$ ps -Af | grep java | grep -o server[0-2]
server1
server2
```

9. Describe the `replicated-topic` topic to see the effect of killing Broker 0.

```
$ kafka-topics \
--zookeeper localhost:2181 \
--describe \
--topic replicated-topic
Topic:replicated-topic  PartitionCount:6    ReplicationFactor:2 Configs:
    Topic: replicated-topic Partition: 0    Leader: -1  Replicas: 0,1
Isr:
    Topic: replicated-topic Partition: 1    Leader: 2   Replicas: 1,2
Isr: 2
    Topic: replicated-topic Partition: 2    Leader: 2   Replicas: 2,0
Isr: 2
    Topic: replicated-topic Partition: 3    Leader: 2   Replicas: 0,2
Isr: 2
    Topic: replicated-topic Partition: 4    Leader: -1  Replicas: 1,0
Isr:
    Topic: replicated-topic Partition: 5    Leader: 2   Replicas: 2,1
Isr: 2
```

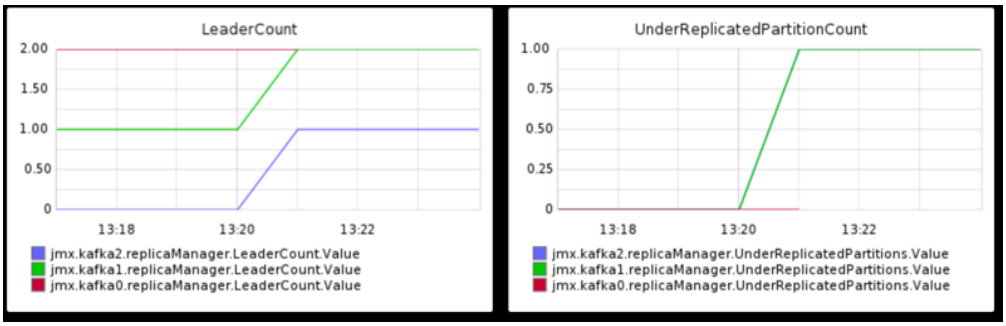
10. Review the server log for Broker 0 (`/var/log/kafka0/server.log`). Look for confirmation of the controlled shutdown succeeding.

```
$ tail /var/log/kafka0/server.log

[2016-04-01 16:47:36,557] INFO Shutdown complete. (kafka.log.LogManager)
[2016-04-01 16:47:36,557] INFO [GroupCoordinator 0]: Shutting down.
(kafka.coordinator.GroupCoordinator)
[2016-04-01 16:47:36,557] INFO [ExpirationReaper-0], Shutting down
(kafka.server.DelayedOperationPurgatory$ExpiredOperationReaper)
[2016-04-01 16:47:36,591] INFO [ExpirationReaper-0], Stopped
(kafka.server.DelayedOperationPurgatory$ExpiredOperationReaper)
[2016-04-01 16:47:36,591] INFO [ExpirationReaper-0], Shutdown completed
(kafka.server.DelayedOperationPurgatory$ExpiredOperationReaper)
[2016-04-01 16:47:36,591] INFO [ExpirationReaper-0], Shutting down
(kafka.server.DelayedOperationPurgatory$ExpiredOperationReaper)
[2016-04-01 16:47:36,791] INFO [ExpirationReaper-0], Stopped
(kafka.server.DelayedOperationPurgatory$ExpiredOperationReaper)
[2016-04-01 16:47:36,791] INFO [ExpirationReaper-0], Shutdown completed
(kafka.server.DelayedOperationPurgatory$ExpiredOperationReaper)
[2016-04-01 16:47:36,792] INFO [GroupCoordinator 0]: Shutdown complete.
(kafka.coordinator.GroupCoordinator)
[2016-04-01 16:47:36,882] INFO [Kafka Server 0], shut down completed
(kafka.server.KafkaServer)
```

11. Review the controller log for Broker 0 (`/var/log/kafka0/controller.log`). Look for confirmation of the controller role being released.

```
$ tail /var/log/kafka0/controller.log

[2016-04-01 16:05:45,504] INFO [Controller 0]: Controller starting up
(kafka.controller.KafkaController)
[2016-04-01 16:05:45,668] INFO [Controller 0]: Controller startup complete
(kafka.controller.KafkaController)
[2016-04-01 16:47:36,793] DEBUG [Controller 0]: Controller resigning,
broker id 0 (kafka.controller.KafkaController)
[2016-04-01 16:47:36,799] DEBUG [Controller 0]: De-registering
IsrChangeNotificationListener (kafka.controller.KafkaController)
[2016-04-01 16:47:36,800] INFO [Partition state machine on Controller 0]:
Stopped partition state machine (kafka.controller.PartitionStateMachine)
[2016-04-01 16:47:36,817] INFO [Replica state machine on controller 0]:
Stopped replica state machine (kafka.controller.ReplicaStateMachine)
[2016-04-01 16:47:36,819] INFO [Controller 0]: Broker 0 resigned as the
controller (kafka.controller.KafkaController)
```

12. Wait for three minutes and observe `LeaderCount` (this should move from one Broker to the other) and `UnderReplicatedPartitionCount` (should increase) in the dashboard.



Recovering From a Broker Failure

Recovering from a broker failure is as simple as restarting the broker and letting Kafka rebalance the leaders and partitions.

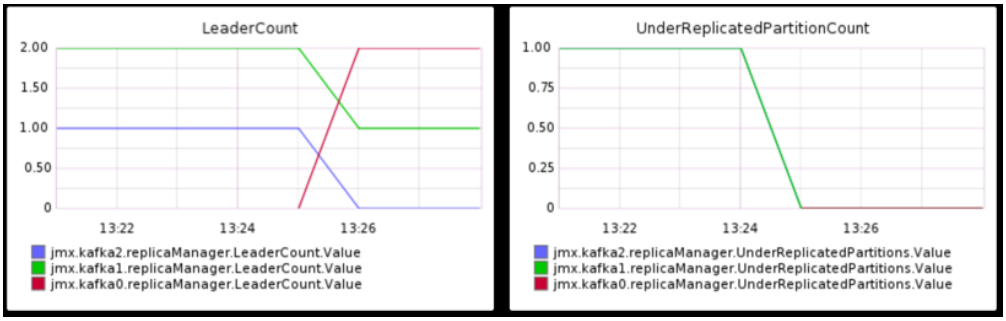
13. Restart Broker 0:

```
$ sudo JMX_PORT=9990 LOG_DIR=/var/log/kafka0 kafka-server-start \
-daemon /vagrant/configfiles/server0.properties
```

14. Verify that three Brokers (`server0`, `server1`, `server2`) are running:

```
$ ps -Af | grep java | grep -o server[0-2]
server0
server1
server2
```

15. Wait for five minutes and observe `LeaderCount` (leader counts should be auto balanced after 5 minutes) and `UnderReplicatedPartitionCount` (should drop to zero for all controllers).



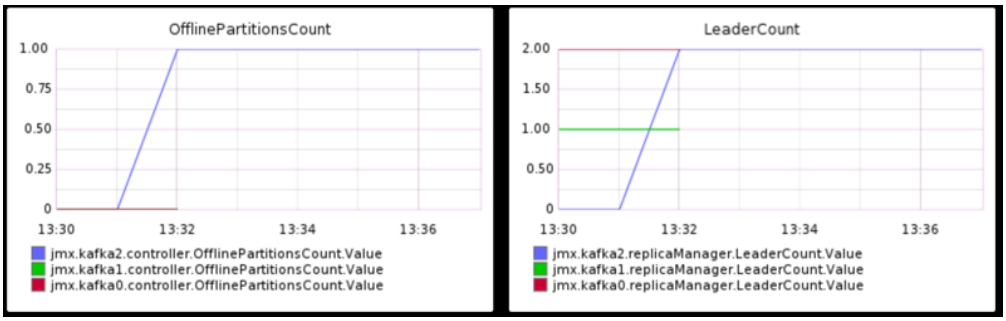
Multiple Broker Failure and Recovery

Now you will simulate a failure of multiple brokers simultaneously and then recover by restarting the Brokers.

16. Terminate two of the brokers on the system and verify that just the third broker is running:

```
$ ps -Af | grep java |grep server0 | tr -s ' ' \
|cut -f2 -d' ' | sudo xargs kill -15

$ ps -Af | grep java |grep server1 | tr -s ' ' \
|cut -f2 -d' '  | sudo xargs kill -15

$ ps -Af | grep java | grep -o server[0-2]
server2
```

17. Observe `OfflinePartitionsCount` (this should increase because we have set replicas to 2 so in some cases both leader and follower partitions will have been lost) and `LeaderCount` (the remaining controller should inherit all leader roles).



18. Restart both Brokers. Wait five minutes for the leaders to rebalance before continuing.

```
$ sudo JMX_PORT=9990 LOG_DIR=/var/log/kafka0 kafka-server-start \
-daemon /vagrant/configfiles/server0.properties

$ sudo JMX_PORT=9991 LOG_DIR=/var/log/kafka1 kafka-server-start \
-daemon /vagrant/configfiles/server1.properties
```

STOP HERE. THIS IS THE END OF THE EXERCISE.

Hands-On Exercise: Oversized Messages

Adding an oversized message

1. Create a new topic called test1, allowing a larger message size.

```
$ kafka-topics \
--zookeeper localhost:2181 \
--create \
--topic test1 \
--partition 1 \
--replication-factor 1 \
--config max.message.bytes=1300000

[2016-04-04 23:02:08,303] WARN

WARNING: you are creating a topic where the the max.message.bytes is
greater than the consumer default. This operation is potentially
dangerous. Consumers will get failures if their fetch.message.max.bytes <
the value you are using.

- value set here: 1300000
- Default Consumer fetch.message.max.bytes: 1048576
- Default Broker max.message.bytes: 1000012

 (kafka.admin.TopicCommand$)
Created topic "test1".
```

2. Produce some text messages to the topic `test1`. Press `Ctrl-c` to return to the command line.

```
$ *kafka-console-producer \
--broker-list localhost:9090 \
--topic test1

Would you
like to
play a game
<Ctrl-c>

*
$
```

3. Consume messages on topic `test1`. Allow the consumer process to run for the duration of the Exercise.

```
$ kafka-console-consumer \
--bootstrap-server localhost:9092 \
--from-beginning \
--topic test1 --new-consumer
Would you
like to
play a game
```

4. Open another terminal window or tab. In this window, add one oversized message. Watch the consumer to note the output.

```
$ kafka-producer-perf-test \
--topic test1 \
--num-records 1 \
--record-size 1200000 \
--throughput 1000 \
--producer-props
bootstrap.servers=localhost:9090,localhost:9091,localhost:9092 \
max.request.size=2000000
```

5. Verify that the console consumer produces an error.

```
ERROR Error processing message, terminating consumer process:
 (kafka.tools.ConsoleConsumer$)
org.apache.kafka.common.errors.RecordTooLargeException: There are some
messages at [Partition=Offset]: {test1-0=3} whose size is larger than the
fetch size 1048576 and hence cannot be ever returned. Increase the fetch
size, or decrease the maximum message size the broker will allow.
```

6. Add the `max.partition.fetch.bytes` property to the `consumer.properties` file.

```
$ printf 'max.partition.fetch.bytes=1300000\n' >> consumer.properties
```

7. Restart the Consumer on topic `test1`.

```
$ kafka-console-consumer \
--consumer.config consumer.properties \
--from-beginning \
--topic test1 \
--bootstrap-server localhost:9092 \
--new-consumer
```

Observe that the oversize message is now displayed. (Note: because of the way it was created, it will appear on the terminal as a large number of non-printable characters.)

8. In a different terminal window, resend the oversize message and verify that it completes without an error. Close the Consumer process when you have verified that the request completes successfully.

```
$ kafka-producer-perf-test \
--topic test1 \
--num-records 1 \
--record-size 1200000 \
--throughput 1000 \
--producer-props
bootstrap.servers=localhost:9090,localhost:9091,localhost:9092 \
max.request.size=2000000
```

If you have more time

9. Create a new topic called `test100` with 1 partition, 2 replicas and allowing a larger message size.

```
$ kafka-topics \
--zookeeper localhost:2181 \
--create \
--topic test100 \
--partition 1 \
--replication-factor 2 \
--config max.message.bytes=1300000

[2016-05-17 14:05:17,164] ERROR

WARNING: you are creating a topic where the max.message.bytes is greater
than the broker default. This operation is dangerous. There are two
potential side effects:
- Consumers will get failures if their fetch.message.max.bytes < the value
you are using
- Producer requests larger than replica.fetch.max.bytes will not replicate
and hence have a higher risk of data loss.
You should ensure both of these settings are greater than the value set
here before using this topic.

- value set here: 1300000
- Default Broker replica.fetch.max.bytes: 1048576
- Default Broker max.message.bytes: 1000012
- Default Consumer fetch.message.max.bytes: 1048576

(kafka.admin.TopicCommand$)
Are you sure you want to continue? [y/n]
y
Created topic "test100".
```

10. Describe the topic.

```
$ kafka-topics \
--zookeeper localhost:2181 \
--describe \
--topic test100
Topic:test100   PartitionCount:1    ReplicationFactor:2
Configs:max.message.bytes=1300000
    Topic: test100  Partition: 0    Leader: 1   Replicas: 1,2   Isr: 1,2
```

11. Produce some text messages to the topic `test100`. Press `Ctrl-c` to return to the command line.

```
$ kafka-console-producer \
--broker-list localhost:9090 \
--topic test100
Greetings programs
Welcome
to the
Grid
<Ctrl-c>
```

12. Start the consumer for the new topic, allowing for oversized messages.

```
$ kafka-console-consumer \
--consumer.config consumer.properties \
--from-beginning \
--topic test100 \
--bootstrap-server localhost:9092 \
--new-consumer
Greetings programs
Welcome
to the
grid
```

13. In a different terminal window, send the oversize message and verify that it completes without an error.

```
$ kafka-producer-perf-test \
--topic test100 \
--num-records 1 \
--record-size 1200000 \
--throughput 1000 \
--producer-props
bootstrap.servers=localhost:9090,localhost:9091,localhost:9092 \
max.request.size=2000000
```

14. Check the status of the topic. Is anything unusual with the replicas?

```
$ kafka-topics \
--zookeeper localhost:2181 \
--describe \
--topic test100
Topic:test100   PartitionCount:1
    ReplicationFactor:2
    Configs:max.message.bytes=1300000
    Topic: test100  Partition: 0    Leader: 1   Replicas: 1,2   Isr: 1
```

15. Examine the `server.log` file on the broker that is not healthy.

```
$ tail /var/log/kafka2/server.log
...
[2016-05-17 14:27:14,759] ERROR [ReplicaFetcherThread-0-1], Replication is
failing due to a message that is greater than replica.fetch.max.bytes.
This generally occurs when the max.message.bytes has been overridden to
exceed this value and a suitably large message has also been sent. To fix
this problem increase replica.fetch.max.bytes in your broker config to be
equal or larger than your settings for max.message.bytes, both at a broker
and topic level. (kafka.server.ReplicaFetcherThread)
```

16. Modify the configuration file for the affected broker and restart it.

```
$ sudo sh -c 'printf replica.fetch.max.bytes=1300000\
>> /vagrant/configfiles/server2.properties'

$ sudo ps -Af |grep java |grep server2 | tr -s ' ' \
|cut -f2 -d' ' | sudo xargs kill -15

$ sudo JMX_PORT=9992 LOG_DIR=/var/log/kafka2 kafka-server-start \
-daemon /vagrant/configfiles/server2.properties
```

17. Verify that the ISR list is now complete.

```
$ kafka-topics \
--zookeeper localhost:2181 \
--describe \
--topic test100
Topic:test100   PartitionCount:1
    ReplicationFactor:2
    Configs:max.message.bytes=1300000
    Topic: test100  Partition: 0    Leader: 0   Replicas: 0,2   Isr: 0,2
```

STOP HERE. THIS IS THE END OF THE EXERCISE.

Hands-On Exercise: Kafka Administration Tools

Increasing the Number of Partitions in a Topic

1. Increase the number of partitions in the topic `test`.

```
$ kafka-topics \
--zookeeper localhost:2181 \
--describe \
--topic test

Topic:test  PartitionCount:6    ReplicationFactor:3 Configs:
    Topic: test Partition: 0    Leader: 1   Replicas: 1,2,0 Isr: 1,0,2
    Topic: test Partition: 1    Leader: 2   Replicas: 2,0,1 Isr: 0,1,2
    Topic: test Partition: 2    Leader: 0   Replicas: 0,1,2 Isr: 0,1,2
    Topic: test Partition: 3    Leader: 1   Replicas: 1,0,2 Isr: 1,0,2
    Topic: test Partition: 4    Leader: 2   Replicas: 2,1,0 Isr: 1,0,2
    Topic: test Partition: 5    Leader: 0   Replicas: 0,2,1 Isr: 0,1,2

$ kafka-topics \
--zookeeper localhost:2181 \
--alter \
--topic test \
--partitions 12

WARNING: If partitions are increased for a topic that has a key, the
partition logic or ordering of the messages will be affected
Adding partitions succeeded!

$ kafka-topics \
--zookeeper localhost:2181 \
--describe \
--topic test

Topic:test  PartitionCount:12   ReplicationFactor:3 Configs:
    Topic: test Partition: 0    Leader: 1   Replicas: 1,2,0 Isr: 1,0,2
    Topic: test Partition: 1    Leader: 2   Replicas: 2,0,1 Isr: 0,1,2
    Topic: test Partition: 2    Leader: 0   Replicas: 0,1,2 Isr: 0,1,2
    Topic: test Partition: 3    Leader: 1   Replicas: 1,0,2 Isr: 1,0,2
    Topic: test Partition: 4    Leader: 2   Replicas: 2,1,0 Isr: 1,0,2
    Topic: test Partition: 5    Leader: 0   Replicas: 0,2,1 Isr: 0,1,2
    Topic: test Partition: 6    Leader: 2   Replicas: 2,1,0 Isr: 2,1,0
    Topic: test Partition: 7    Leader: 0   Replicas: 0,2,1 Isr: 0,2,1
    Topic: test Partition: 8    Leader: 1   Replicas: 1,0,2 Isr: 1,0,2
    Topic: test Partition: 9    Leader: 2   Replicas: 2,0,1 Isr: 2,0,1
    Topic: test Partition: 10   Leader: 0   Replicas: 0,1,2 Isr: 0,1,2
    Topic: test Partition: 11   Leader: 1   Replicas: 1,2,0 Isr: 1,2,0
```

Checking Consumer Offsets

2. View the Consumer offsets with the following command:

```
$ kafka-console-consumer \
--consumer.config consumer.properties \
--from-beginning \
--topic new-topic \
--bootstrap-server localhost:9090 \
--new-consumer
```

Leave this Consumer running for the duration of the Exercise.

3. In another terminal window, run the following command:

```
$ kafka-consumer-groups \
--bootstrap-server localhost:9092 \
--group test-consumer-group \
--describe \
--new-consumer
```

Making Topic-Level Configuration Changes

4. Add a new configuration value:

```
$ kafka-configs \
--zookeeper localhost:2181 \
--alter \
--entity-type topics \
--entity-name i-love-logs \
--alter \
--add-config segment.bytes=1000000

Updated config for topic: "i-love-logs"
```

5. Add some more messages to the topic:

```
$ kafka-producer-perf-test \
--topic i-love-logs \
--num-records 10000000 \
--record-size 100 \
--throughput 1000 \
--producer-props
bootstrap.servers=localhost:9090,localhost:9091,localhost:9092
```

6. Now observe the log segment file size under `/var/lib/kafka_?/`
7. Delete a configuration value:

```
$ kafka-configs --zookeeper localhost:2181 \
--alter --entity-type topics \
--entity-name i-love-logs \
--alter --delete-config segment.bytes

Updated config for topic: "i-love-logs"
```

Deleting Topics in the Cluster

If you need to free up space in your environment, you can delete topics.

8. Stop all Brokers.

```
$ ps -Af |grep java |grep server0 | tr -s ' ' \
|cut -f2 -d' ' | sudo xargs kill -15

$ ps -Af |grep java |grep server1 | tr -s ' ' \
|cut -f2 -d' ' | sudo xargs kill -15

$ ps -Af |grep java |grep server2 | tr -s ' ' \
|cut -f2 -d' ' | sudo xargs kill -15
```

9. Add the following new property to the property file of all Brokers (`/vagrant/configfiles/server?.properties`).

```
delete.topic.enable=true
```

10. Restart all Brokers.

```
$ sudo JMX_PORT=9990 LOG_DIR=/var/log/kafka0 kafka-server-start \
-daemon /vagrant/configfiles/server0.properties

$ sudo JMX_PORT=9991 LOG_DIR=/var/log/kafka1 kafka-server-start \
-daemon /vagrant/configfiles/server1.properties

$ sudo JMX_PORT=9992 LOG_DIR=/var/log/kafka2 kafka-server-start \
-daemon /vagrant/configfiles/server2.properties
```

11. Verify that three Brokers (`server0`, `server1`, `server2`) are running:

```
$ ps -Af | grep java | grep -o server[0-2]
server0
server1
server2
```

12. If they are currently running, stop the Producer and the Consumer before deleting the topic. Otherwise, the topic will be recreated automatically. Once they have been stopped, delete the topic.

```
$ kafka-topics \
--zookeeper localhost:2181 \
--list

consumer_offsets
_schemas
i-love-logs
new-topic
test
test1
test2
test3

$ kafka-topics \
--zookeeper localhost:2181 \
--delete \
--topic test

Topic test is marked for deletion.
Note: This will have no impact if delete.topic.enable
is not set to true.

$ kafka-topics \
--zookeeper localhost:2181 \
--list

consumer_offsets
_schemas
i-love-logs
new-topic
test1
test2
test3
```

Reassigning Partitions in a Topic

13. Create a new topic with 6 partitions and 2 replicas, on only Broker 0 and Broker 1.

```
$ kafka-topics \
--zookeeper localhost:2181 \
--create \
--topic test2 \
--replica-assignment 0:1,1:0,0:1,1:0,0:1,1:0

Created topic "test2".
```

14. Produce 2GB of data to topic `test2`. (If you have enough disk space, change `--num-records` to `20000000` to load 20GB of data so that you have more time to observe the partition reassignment process.) Note: wait until this command has completed before continuing.

```
$ kafka-producer-perf-test \
--topic test2 \
--num-records 2000000 \
--record-size 1000 \
--throughput 1000000000 \
--producer-props
bootstrap.servers=localhost:9090,localhost:9091,localhost:9092
```

15. In a different terminal window, start a Consumer on topic `test2`.

```
$ kafka-consumer-perf-test \
--broker-list localhost:9090 \
--topic test2 \
--group test-group \
--threads 1 \
--show-detailed-stats \
--reporting-interval 5000 \
--messages 10000000 \
--new-consumer
```

16. Open a third terminal window or tab. Prepare the `topics-to-move` file.

```
$ printf \
'{"topics": [{"topic": "test2"}],"version":1}\n' \
> topics-to-move.json
```

17. Generate the reassignment plan.

```
$ kafka-reassign-partitions \
--zookeeper localhost:2181 \
--topics-to-move-json-file topics-to-move.json \
--broker-list "0,1,2"-generate > reassignment.json

$ cat reassignment.json

Current partition replica assignment
{"version":1,"partitions":[{"topic":"test2","partition":0,"replicas":[0,1]
},{"topic":"test2","partition":2,"replicas":[0,1]},{"topic":"test2","parti
tion":4,"replicas":[0,1]},{"topic":"test2","partition":1,"replicas":[1,0]}
,{"topic":"test2","partition":5,"replicas":[1,0]},{"topic":"test2","partit
ion":3,"replicas":[1,0]}]}
Proposed partition reassignment configuration
{"version":1,"partitions":[{"topic":"test2","partition":0,"replicas":[2,0]
},{"topic":"test2","partition":2,"replicas":[1,2]},{"topic":"test2","parti
tion":4,"replicas":[0,2]},{"topic":"test2","partition":1,"replicas":[0,1]}
,{"topic":"test2","partition":5,"replicas":[1,0]},{"topic":"test2","partit
ion":3,"replicas":[2,1]}]}
```

Edit the `reassignment.json` file so it only includes the proposed partition assignment configuration in JSON (i.e., just the last line).

18. Execute the reassignment plan.

```
$ kafka-reassign-partitions \
--zookeeper localhost:2181 \
--reassignment-json-file reassignment.json \
--execute

Current partition replica assignment
{"version":1,"partitions":[{"topic":"test2","partition":0,"replicas":[0,1]
},{"topic":"test2","partition":2,"replicas":[0,1]},{"topic":"test2","parti
tion":4,"replicas":[0,1]},{"topic":"test2","partition":1,"replicas":[1,0]}
,{"topic":"test2","partition":5,"replicas":[1,0]},{"topic":"test2","partit
ion":3,"replicas":[1,0]}]}
Save this to use as the --reassignment-json-file option during rollback
Successfully started reassignment of partitions
{"version":1,"partitions":[{"topic":"test2","partition":0,"replicas":[2,0]
},{"topic":"test2","partition":2,"replicas":[1,2]},{"topic":"test2","parti
tion":4,"replicas":[0,2]},{"topic":"test2","partition":1,"replicas":[0,1]}
,{"topic":"test2","partition":5,"replicas":[1,0]},{"topic":"test2","partit
ion":3,"replicas":[2,1]}]
```

19. Observe if the reassignment has completed.

```
$ kafka-topics \
--zookeeper localhost:2181 \
--describe \
--topic test2

Topic:test2 PartitionCount:6    ReplicationFactor:2 Configs:
    Topic: test2    Partition: 0    Leader: 1   Replicas: 1,2   Isr: 1,2
    Topic: test2    Partition: 1    Leader: 2   Replicas: 2,0   Isr: 0,2
    Topic: test2    Partition: 2    Leader: 0   Replicas: 0,1   Isr: 0,1
    Topic: test2    Partition: 3    Leader: 1   Replicas: 1,0   Isr: 1,0
    Topic: test2    Partition: 4    Leader: 2   Replicas: 2,1   Isr: 1,2
    Topic: test2    Partition: 5    Leader: 0   Replicas: 0,2   Isr: 0,2
```

You can also verify the process whilst it is running.

```
$ kafka-reassign-partitions \
--zookeeper localhost:2181 \
--reassignment-json-file reassignment.json \
--verify

Status of partition reassignment:
Reassignment of partition [test2,3] is still in progress
Reassignment of partition [test2,1] completed successfully
Reassignment of partition [test2,5] completed successfully
Reassignment of partition [test2,2] completed successfully
Reassignment of partition [test2,0] completed successfully
Reassignment of partition [test2,4] completed successfully
```

[start=20] Stop the producers and consumers you started for this exercise, if necessary.

Simulate a Completely Failed Broker

In previous Exercises, the simulated failures were situations where the Broker could be restarted to recover from the fault. Now you will simulate a completely failed broker which requires a replacement system.

21. Stop Broker 0. Verify that just two of the brokers are running.

```
$ ps -Af |grep java |grep server0 | tr -s ' ' \
|cut -f2 -d' ' | sudo xargs kill -15

$ ps -Af | grep java | grep -o server[0-2]
server1
server2
```

22. Remove the logs for Broker 0.

```
$ ls /var/lib/kafka_0

i-love-logs-1    recovery-point-offset-checkpoint  _schemas-0
meta.properties  replication-offset-checkpoint

$ sudo rm -rf /var/lib/kafka_0
$ ls /var/lib/kafka_0

$
```

23. Restart Broker 0.

```
$ sudo JMX_PORT=9990 LOG_DIR=/var/log/kafka0 kafka-server-start \
-daemon /vagrant/configfiles/server0.properties
```

24. Verify that three Brokers (`server0`, `server1`, `server2`) are running:

```
$ ps -Af | grep java | grep -o server[0-2]
server0
server1
server2
```

25. Look at the log (the data should be re-replicated back).

```
$ ls /var/lib/kafka_0

i-love-logs-1    recovery-point-offset-checkpoint  _schemas-0
meta.properties  replication-offset-checkpoint
```

26. Return to the terminal session with the producer process and press `Ctrl-c` to end the process.

STOP HERE. THIS IS THE END OF THE EXERCISE.