1
Real-Time Processing and Storm Introduction

With the exponential growth in the amount of data being generated and advanced data-capturing capabilities, enterprises are facing the challenge of making sense out of this mountain of raw data. On the batch processing front, Hadoop has emerged as the go-to framework to deal with big data. Until recently, there has been a void when one looks for frameworks to build real-time stream processing applications. Such applications have become an integral part of a lot of businesses as they enable them to respond swiftly to events and adapt to changing situations. Examples of this are monitoring social media to analyze public response to any new product that you launch and predicting the outcome of an election based on the sentiments of election-related posts.

Organizations are collecting a large volume of data from external sources and want to evaluate/process the data in real time to get market trends, detect fraud, identify user behavior, and so on. The need for real-time processing is increasing day by day and we require a real-time system/platform that should support the following features:

- **Scalable**: The platform should be horizontally scalable without any down time.
- **Fault tolerance**: The platform should be able to process the data even after some of the nodes in a cluster go down.
- **No data lost**: The platform should provide the guaranteed processing of messages.
- **High throughput**: The system should be able to support millions of records per second and also support any size of messages.

- **Easy to operate**: The system should have easy installation and operation. Also, the expansion of clusters should be an easy process.
- **Multiple languages**: The platform should support multiple languages. The end user should be able to write code in different languages. For example, a user can write code in Python, Scala, Java, and so on. Also, we can execute different language code inside the one cluster.
- **Cluster isolation**: The system should support isolation so that dedicated processes can be assigned to dedicated machines for processing.

Apache Storm

Apache Storm has emerged as the platform of choice for industry leaders to develop distributed, real-time, data processing platforms. It provides a set of primitives that can be used to develop applications that can process a very large amount of data in real time in a highly scalable manner.

Storm is to real-time processing what Hadoop is to batch processing. It is open source software, and managed by Apache Software Foundation. It has been deployed to meet real-time processing needs by companies such as Twitter, Yahoo!, and Flipboard. Storm was first developed by Nathan Marz at BackType, a company that provided social search applications. Later, BackType was acquired by Twitter, and it is a critical part of their infrastructure. Storm can be used for the following use cases:

- **Stream processing**: Storm is used to process a stream of data and update a variety of databases in real time. This processing occurs in real time and the processing speed needs to match the input data speed.
- **Continuous computation**: Storm can do continuous computation on data streams and stream the results to clients in real time. This might require processing each message as it comes in or creating small batches over a short time. An example of continuous computation is streaming trending topics on Twitter into browsers.
- **Distributed RPC**: Storm can parallelize an intense query so that you can compute it in real time.
- **Real-time analytics**: Storm can analyze and respond to data that comes from different data sources as they happen in real time.

In this chapter, we will cover the following topics:

- What is a Storm?
- Features of Storm
- Architecture and components of a Storm cluster
- Terminologies of Storm
- Programming language
- Operation modes

Features of Storm

The following are some of the features of Storm that make it a perfect solution to process streams of data in real time:

- **Fast**: Storm has been reported to process up to 1 million tuples/records per second per node.
- **Horizontally scalable**: Being fast is a necessary feature to build a high volume/velocity data processing platform, but a single node will have an upper limit on the number of events that it can process per second. A node represents a single machine in your setup that executes Storm applications. Storm, being a distributed platform, allows you to add more nodes to your Storm cluster and increase the processing capacity of your application. Also, it is linearly scalable, which means that you can double the processing capacity by doubling the nodes.
- **Fault tolerant**: Units of work are executed by worker processes in a Storm cluster. When a worker dies, Storm will restart that worker, and if the node on which the worker is running dies, Storm will restart that worker on some other node in the cluster. This feature will be covered in more detail in `Chapter 3`, *Storm Parallelism and Data Partitioning*.
- **Guaranteed data processing**: Storm provides strong guarantees that each message entering a Storm process will be processed at least once. In the event of failures, Storm will replay the lost tuples/records. Also, it can be configured so that each message will be processed only once.
- **Easy to operate**: Storm is simple to deploy and manage. Once the cluster is deployed, it requires little maintenance.
- **Programming language agnostic**: Even though the Storm platform runs on **Java virtual machine** (**JVM**), the applications that run over it can be written in any programming language that can read and write to standard input and output streams.

Storm components

A Storm cluster follows a master-slave model where the master and slave processes are coordinated through ZooKeeper. The following are the components of a Storm cluster.

Nimbus

The Nimbus node is the master in a Storm cluster. It is responsible for distributing the application code across various worker nodes, assigning tasks to different machines, monitoring tasks for any failures, and restarting them as and when required.

Nimbus is stateless and stores all of its data in ZooKeeper. There is a single Nimbus node in a Storm cluster. If the active node goes down, then the passive node will become an Active node. It is designed to be fail-fast, so when the active Nimbus dies, the passive node will become an active node, or the down node can be restarted without having any effect on the tasks already running on the worker nodes. This is unlike Hadoop, where if the JobTracker dies, all the running jobs are left in an inconsistent state and need to be executed again. The Storm workers can work smoothly even if all the Nimbus nodes go down but the user can't submit any new jobs into the cluster or the cluster will not be able to reassign the failed workers to another node.

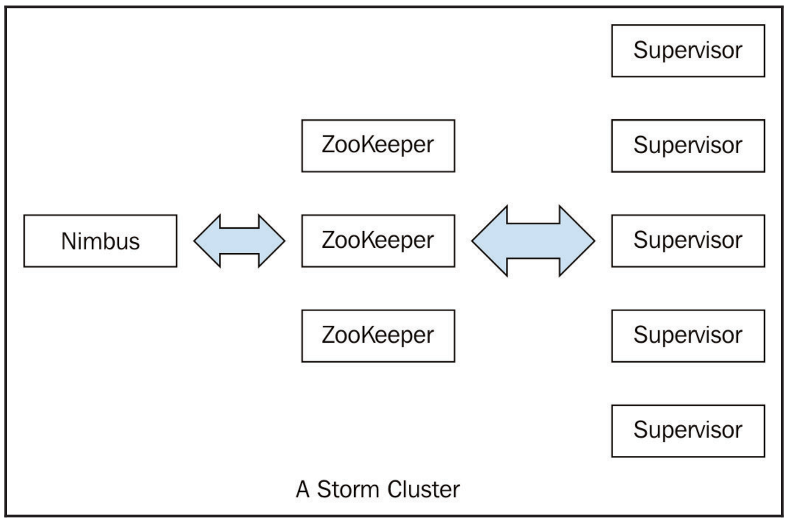
Supervisor nodes

Supervisor nodes are the worker nodes in a Storm cluster. Each supervisor node runs a supervisor daemon that is responsible for creating, starting, and stopping worker processes to execute the tasks assigned to that node. Like Nimbus, a supervisor daemon is also fail-fast and stores all of its states in ZooKeeper so that it can be restarted without any state loss. A single supervisor daemon normally handles multiple worker processes running on that machine.

The ZooKeeper cluster

In any distributed application, various processes need to coordinate with each other and share some configuration information. ZooKeeper is an application that provides all these services in a reliable manner. As a distributed application, Storm also uses a ZooKeeper cluster to coordinate various processes. All of the states associated with the cluster and the various tasks submitted to Storm are stored in ZooKeeper. Nimbus and supervisor nodes do not communicate directly with each other, but through ZooKeeper. As all data is stored in ZooKeeper, both Nimbus and the supervisor daemons can be killed abruptly without adversely affecting the cluster.

The following is an architecture diagram of a Storm cluster:



The Storm data model

The basic unit of data that can be processed by a Storm application is called a tuple. Each tuple consists of a predefined list of fields. The value of each field can be a byte, char, integer, long, float, double, Boolean, or byte array. Storm also provides an API to define your own datatypes, which can be serialized as fields in a tuple.

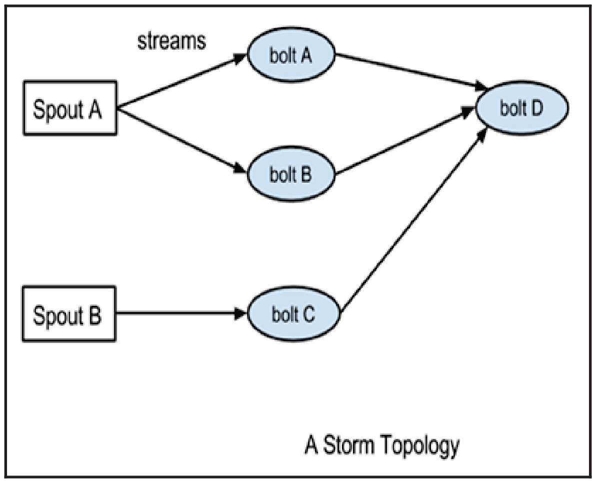
A tuple is dynamically typed, that is, you just need to define the names of the fields in a tuple and not their datatype. The choice of dynamic typing helps to simplify the API and makes it easy to use. Also, since a processing unit in Storm can process multiple types of tuples, it's not practical to declare field types.

Each of the fields in a tuple can be accessed by its name, `getValueByField(String)`, or its positional index, `getValue(int)`, in the tuple. Tuples also provide convenient methods such as `getIntegerByField(String)` that save you from typecasting the objects. For example, if you have a *Fraction (numerator, denominator)* tuple, representing fractional numbers, then you can get the value of the numerator by either using `getIntegerByField("numerator")` or `getInteger(0)`.

You can see the full set of operations supported by `org.apache.storm.tuple.Tuple` in the Java doc that is located at `https://storm.apache.org/releases/1.0.2/javadocs/org/apache/storm/tuple/Tuple.html`.

Definition of a Storm topology

In Storm terminology, a topology is an abstraction that defines the graph of the computation. You create a Storm topology and deploy it on a Storm cluster to process data. A topology can be represented by a direct acyclic graph, where each node does some kind of processing and forwards it to the next node(s) in the flow. The following diagram is a sample Storm topology:



The following are the components of a Storm topology:

- **Tuple**: A single message/record that flows between the different instances of a topology is called a tuple.
- **Stream**: The key abstraction in Storm is that of a stream. A stream is an unbounded sequence of tuples that can be processed in parallel by Storm. Each stream can be processed by a single or multiple types of bolts (the processing units in Storm, which are defined later in this section). Thus, Storm can also be viewed as a platform to transform streams. In the preceding diagram, streams are represented by arrows. Each stream in a Storm application is given an ID and the bolts can produce and consume tuples from these streams on the basis of their ID. Each stream also has an associated schema for the tuples that will flow through it.
- **Spout**: A spout is the source of tuples in a Storm topology. It is responsible for reading or listening to data from an external source, for example, by reading from a log file or listening for new messages in a queue and publishing them--emitting in Storm terminology into streams. A spout can emit multiple streams, each of a different schema. For example, it can read records of 10 fields from a log file and emit them as different streams of seven-fields tuples and four-fields tuples each.

  The `org.apache.storm.spout.ISpout` interface is the interface used to define spouts. If you are writing your topology in Java, then you should use `org.apache.storm.topology.IRichSpout` as it declares methods to use with the `TopologyBuilder` API. Whenever a spout emits a tuple, Storm tracks all the tuples generated while processing this tuple, and when the execution of all the tuples in the graph of this source tuple is complete, it will send an acknowledgement back to the spout. This tracking happens only if a message ID was provided when emitting the tuple. If null was used as the message ID, this tracking will not happen.

  A tuple processing timeout can also be defined for a topology, and if a tuple is not processed within the specified timeout, a fail message will be sent back to the spout. Again, this will happen only if you define a message ID. A small performance gain can be extracted out of Storm at the risk of some data loss by disabling the message acknowledgements, which can be done by skipping the message ID while emitting tuples.

The important methods of spout are:

- `nextTuple()`: This method is called by Storm to get the next tuple from the input source. Inside this method, you will have the logic of reading data from external sources and emitting them to an instance of `org.apache.storm.spout.ISpoutOutputCollector`. The schema for streams can be declared by using the `declareStream` method of `org.apache.storm.topology.OutputFieldsDeclarer`.

  If a spout wants to emit data to more than one stream, it can declare multiple streams using the `declareStream` method and specify a stream ID while emitting the tuple. If there are no more tuples to emit at the moment, this method will not be blocked. Also, if this method does not emit a tuple, then Storm will wait for 1 millisecond before calling it again. This waiting time can be configured using the `topology.sleep.spout.wait.strategy.time.ms` setting.

- `ack(Object msgId)`: This method is invoked by Storm when the tuple with the given message ID is completely processed by the topology. At this point, the user should mark the message as processed and do the required cleaning up, such as removing the message from the message queue so that it does not get processed again.
- `fail(Object msgId)`: This method is invoked by Storm when it identifies that the tuple with the given message ID has not been processed successfully or has timed out of the configured interval. In such scenarios, the user should do the required processing so that the messages can be emitted again by the `nextTuple` method. A common way to do this is to put the message back in the incoming message queue.
- `open()`: This method is called only once--when the spout is initialized. If it is required to connect to an external source for the input data, define the logic to connect to the external source in the open method, and then keep fetching the data from this external source in the `nextTuple` method to emit it further.

Another point to note while writing your spout is that none of the methods should be blocking, as Storm calls all the methods in the same thread. Every spout has an internal buffer to keep track of the status of the tuples emitted so far. The spout will keep the tuples in this buffer until they are either acknowledged or failed, calling the `ack` or `fail` method, respectively. Storm will call the `nextTuple` method only when this buffer is not full.

- **Bolt**: A bolt is the processing powerhouse of a Storm topology and is responsible for transforming a stream. Ideally, each bolt in the topology should be doing a simple transformation of the tuples, and many such bolts can coordinate with each other to exhibit a complex transformation.

  The `org.apache.storm.task.IBolt` interface is preferably used to define bolts, and if a topology is written in Java, you should use the `org.apache.storm.topology.IRichBolt` interface. A bolt can subscribe to multiple streams of other components--either spouts or other bolts--in the topology and similarly can emit output to multiple streams. Output streams can be declared using the `declareStream` method of `org.apache.storm.topology.OutputFieldsDeclarer`.

  The important methods of a bolt are:

  - `execute(Tuple input)`: This method is executed for each tuple that comes through the subscribed input streams. In this method, you can do whatever processing is required for the tuple and then produce the output either in the form of emitting more tuples to the declared output streams, or other things such as persisting the results in a database.

    You are not required to process the tuple as soon as this method is called, and the tuples can be held until required. For example, while joining two streams, when a tuple arrives you can hold it until its counterpart also comes, and then you can emit the joined tuple.

The metadata associated with the tuple can be retrieved by the various methods defined in the `Tuple` interface. If a message ID is associated with a tuple, the execute method must publish an `ack` or `fail` event using `OutputCollector` for the bolt, or else Storm will not know whether the tuple was processed successfully. The `org.apache.storm.topology.IBasicBolt` interface is a convenient interface that sends an acknowledgement automatically after the completion of the execute method. If a fail event is to be sent, this method should throw `org.apache.storm.topology.FailedException`.

- `prepare(Map stormConf, TopologyContext context, OutputCollector collector)`: A bolt can be executed by multiple workers in a Storm topology. The instance of a bolt is created on the client machine and then serialized and submitted to Nimbus. When Nimbus creates the worker instances for the topology, it sends this serialized bolt to the workers. The work will desterilize the bolt and call the `prepare` method. In this method, you should make sure the bolt is properly configured to execute tuples. Any state that you want to maintain can be stored as instance variables for the bolt that can be serialized/deserialized later.

Operation modes in Storm

Operation modes indicate how the topology is deployed in Storm. Storm supports two types of operation modes to execute the Storm topology:

- **Local mode**: In local mode, Storm topologies run on the local machine in a single JVM. This mode simulates a Storm cluster in a single JVM and is used for the testing and debugging of a topology.
- **Remote mode**: In remote mode, we will use the Storm client to submit the topology to the master along with all the necessary code required to execute the topology. Nimbus will then take care of distributing your code.

In the next chapter, we are going to cover both local and remote mode in more detail, along with a sample example.

Programming languages

Storm was designed from the ground up to be usable with any programming language. At the core of Storm is a thrift definition for defining and submitting topologies. Since thrift can be used in any language, topologies can be defined and submitted in any language.

Similarly, spouts and bolts can be defined in any language. Non-JVM spouts and bolts communicate with Storm over a JSON-based protocol over `stdin`/`stdout`. Adapters that implement this protocol exist for Ruby, Python, JavaScript, and Perl. You can refer to `https://github.com/apache/storm/tree/master/storm-multilang` to find out about the implementation of these adapters.

Storm-starter has an example topology, `https://github.com/apache/storm/tree/master/examples/storm-starter/multilang/resources`, which implements one of the bolts in Python.

Summary

In this chapter, we introduced you to the basics of Storm and the various components that make up a Storm cluster. We saw a definition of different deployment/operation modes in which a Storm cluster can operate.

In the next chapter, we will set up a single and three-node Storm cluster and see how we can deploy the topology on a Storm cluster. We will also see different types of stream groupings supported by Storm and the guaranteed message semantic provided by Storm.

2
Storm Deployment, Topology Development, and Topology Options

In this chapter, we are going to start with deployment of Storm on multiple node (three Storm and three ZooKeeper) clusters. This chapter is very important because it focuses on how we can set up the production Storm cluster and why we need the high availability of both the Storm Supervisor, Nimbus, and ZooKeeper (as Storm uses ZooKeeper for storing the metadata of the cluster, topology, and so on)?

The following are the key points that we are going to cover in this chapter:

- Deployment of the Storm cluster
- Program and deploy the word count example
- Different options of the Storm UI--kill, active, inactive, and rebalance
- Walkthrough of the Storm UI
- Dynamic log level settings
- Validating the Nimbus high availability

Storm prerequisites

You should have the Java JDK and ZooKeeper ensemble installed before starting the deployment of the Storm cluster.

Installing Java SDK 7

Perform the following steps to install the Java SDK 7 on your machine. You can also go with JDK 1.8:

1. Download the Java SDK 7 RPM from Oracle's site (`http://www.oracle.com/technetwork/java/javase/downloads/index.html`).
2. Install the Java `jdk-7u<version>-linux-x64.rpm` file on your CentOS machine using the following command:

   ```
   sudo rpm -ivh jdk-7u<version>-linux-x64.rpm
   ```

3. Add the following environment variable in the `~/.bashrc` file:

   ```
   export JAVA_HOME=/usr/java/jdk<version>
   ```

4. Add the path of the `bin` directory of the JDK to the `PATH` system environment variable to the `~/.bashrc` file:

   ```
   export PATH=$JAVA_HOME/bin:$PATH
   ```

5. Run the following command to reload the `bashrc` file on the current login terminal:

   ```
   source ~/.bashrc
   ```

6. Check the Java installation as follows:

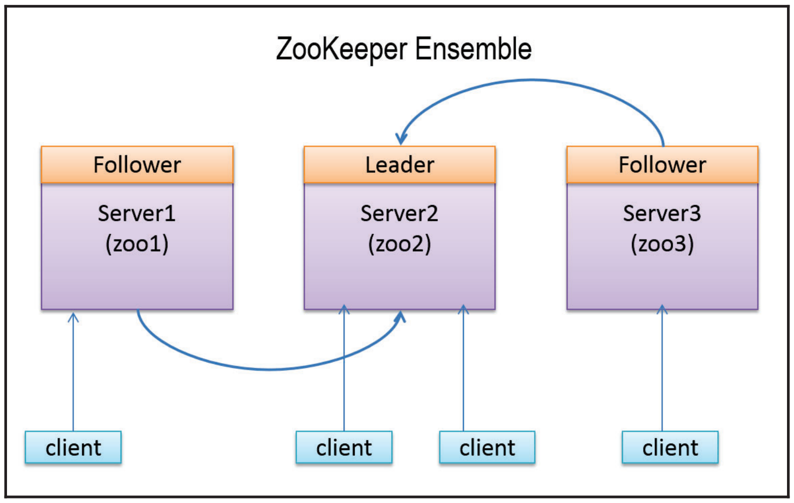
   ```
   java -version
   ```

   The output of the preceding command is as follows:

   ```
   java version "1.7.0_71"
   Java(TM) SE Runtime Environment (build 1.7.0_71-b14)
   Java HotSpot(TM) 64-Bit Server VM (build 24.71-b01, mixed mode)
   ```

Deployment of the ZooKeeper cluster

In any distributed application, various processes need to coordinate with each other and share configuration information. ZooKeeper is an application that provides all these services in a reliable manner. Being a distributed application, Storm also uses a ZooKeeper cluster to coordinate various processes. All of the states associated with the cluster and the various tasks submitted to Storm are stored in ZooKeeper. This section describes how you can set up a ZooKeeper cluster. We will be deploying a ZooKeeper ensemble of three nodes that will handle one node failure. Following is the deployment diagram of the three node ZooKeeper ensemble:



In the ZooKeeper ensemble, one node in the cluster acts as the leader, while the rest are followers. If the leader node of the ZooKeeper cluster dies, then an election for the new leader takes places among the remaining live nodes, and a new leader is elected. All write requests coming from clients are forwarded to the leader node, while the follower nodes only handle the read requests. Also, we can't increase the write performance of the ZooKeeper ensemble by increasing the number of nodes because all write operations go through the leader node.

It is advised to run an odd number of ZooKeeper nodes, as the ZooKeeper cluster keeps working as long as the majority (the number of live nodes is greater than *n*/2, where *n* being the number of deployed nodes) of the nodes are running. So if we have a cluster of four ZooKeeper nodes (*3* > *4*/*2*; only one node can die), then we can handle only one node failure, while if we had five nodes (*3* > *5*/*2*; two nodes can die) in the cluster, then we can handle two node failures.

Steps 1 to 4 need to be performed on each node to deploy the ZooKeeper ensemble:

1. Download the latest stable ZooKeeper release from the ZooKeeper site (`http://zookeeper.apache.org/releases.html`). At the time of writing, the latest version is ZooKeeper 3.4.6.
2. Once you have downloaded the latest version, unzip it. Now, we set up the `ZK_HOME` environment variable to make the setup easier.
3. Point the `ZK_HOME` environment variable to the unzipped directory. Create the configuration file, `zoo.cfg`, at the `$ZK_HOME/conf` directory using the following commands:

   ```
   cd $ZK_HOME/conf
   touch zoo.cfg
   ```

4. Add the following properties to the `zoo.cfg` file:

   ```
   tickTime=2000
   dataDir=/var/zookeeper
   clientPort=2181
   initLimit=5
   syncLimit=2
   server.1=zoo1:2888:3888
   server.2=zoo2:2888:3888
   server.3=zoo3.2888.3888
   ```

   Here, `zoo1`, `zoo2`, and `zoo3` are the IP addresses of the ZooKeeper nodes. The following are the definitions for each of the properties:

   - `tickTime`: This is the basic unit of time in milliseconds used by ZooKeeper. It is used to send heartbeats, and the minimum session timeout will be twice the `tickTime` value.
   - `dataDir`: This is the directory to store the in-memory database snapshots and transactional log.
   - `clientPort`: This is the port used to listen to client connections.
   - `initLimit`: This is the number of `tickTime` values needed to allow followers to connect and sync to a leader node.
   - `syncLimit`: This is the number of `tickTime` values that a follower can take to sync with the leader node. If the sync does not happen within this time, the follower will be dropped from the ensemble.

The last three lines of the `server.id=host:port:port` format specify that there are three nodes in the ensemble. In an ensemble, each ZooKeeper node must have a unique ID number between 1 and 255. This ID is defined by creating a file named `myid` in the `dataDir` directory of each node. For example, the node with the ID 1 (`server.1=zoo1:2888:3888`) will have a `myid` file at directory `/var/zookeeper` with `text 1` inside it.

For this cluster, create the `myid` file at three locations, shown as follows:

```
At zoo1 /var/zookeeper/myid contains 1
At zoo2 /var/zookeeper/myid contains 2
At zoo3 /var/zookeeper/myid contains 3
```

5. Run the following command on each machine to start the ZooKeeper cluster:

```
bin/zkServer.sh start
```

Check the status of the ZooKeeper nodes by performing the following steps:

6. Run the following command on the `zoo1` node to check the first node's status:

```
bin/zkServer.sh status
```

The following information is displayed:

```
JMX enabled by default
Using config: /home/root/zookeeper-3.4.6/bin/../conf/zoo.cfg
Mode: follower
```

The first node is running in `follower` mode.

7. Check the status of the second node by performing the following command:

```
bin/zkServer.sh status
```

The following information is displayed:

```
JMX enabled by default
Using config: /home/root/zookeeper-3.4.6/bin/../conf/zoo.cfg
Mode: leader
```

The second node is running in `leader` mode.

8. Check the status of the third node by performing the following command:

   ```
   bin/zkServer.sh status
   ```

   The following information is displayed:

   ```
   JMX enabled by default
   Using config: /home/root/zookeeper-3.4.6/bin/../conf/zoo.cfg
   Mode: follower
   ```

   The third node is running in `follower` mode.

9. Run the following command on the leader machine to stop the leader node:

```
bin/zkServer.sh stop
```

Now, check the status of the remaining two nodes by performing the following steps:

10. Check the status of the first node using the following command:

```
bin/zkServer.sh status
```

   The following information is displayed:

   ```
   JMX enabled by default
   Using config: /home/root/zookeeper-3.4.6/bin/../conf/zoo.cfg
   Mode: follower
   ```

   The first node is again running in `follower` mode.

11. Check the status of the second node using the following command:

    ```
    bin/zkServer.sh status
    ```

    The following information is displayed:

    ```
    JMX enabled by default
    Using config: /home/root/zookeeper-3.4.6/bin/../conf/zoo.cfg
    Mode: leader
    ```

    The third node is elected as the new leader.

12. Now, restart the third node with the following command:

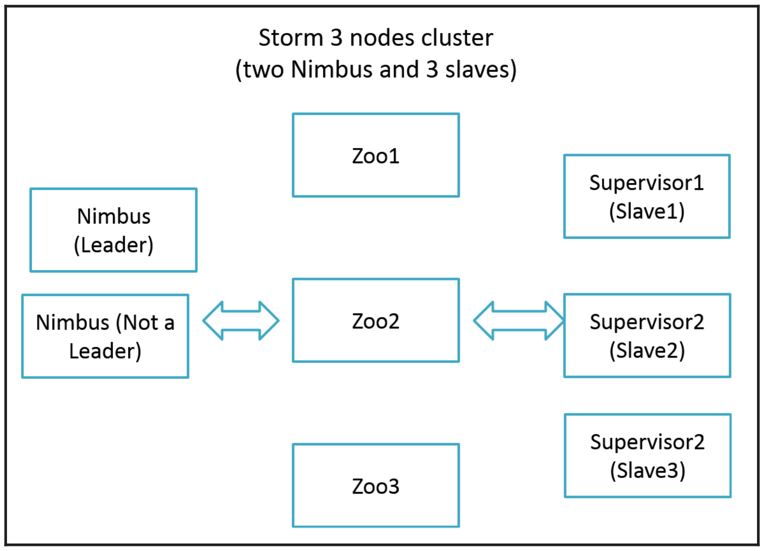
```
bin/zkServer.sh status
```

This was a quick introduction to setting up ZooKeeper that can be used for development; however, it is not suitable for production. For a complete reference on ZooKeeper administration and maintenance, please refer to the online documentation at the ZooKeeper site at `http://zookeeper.apache.org/doc/trunk/zookeeperAdmin.html`.

Setting up the Storm cluster

In this chapter, we will learn how to set up a three nodes Storm cluster, of which one node will be the active master node (Nimbus) and the other two will be worker nodes (supervisors).

The following is the deployment diagram of our three node Storm cluster:



The following are the steps that need to be performed to set up a three node Storm cluster:

1. Install and run the ZooKeeper cluster. The steps for installing ZooKeeper are mentioned in the previous section.
2. Download the latest stable Storm release from `https://storm.apache.org/downloads.html`; at the time of writing, the latest version is Storm 1.0.2.

3. Once you have downloaded the latest version, copy and unzip it in all three machines. Now, we will set the `$STORM_HOME` environment variable on each machine to make the setup easier. The `$STORM_HOME` environment contains the path of the Storm `home` folder (for example, export `STORM_HOME=/home/user/storm-1.0.2`).
4. Go to the `$STORM_HOME/conf` directory in the master nodes and add the following lines to the `storm.yaml` file:

```
storm.zookeeper.servers:
- "zoo1"
- "zoo2"
- "zoo3"
storm.zookeeper.port: 2181
nimbus.seeds: "nimbus1,nimbus2"
storm.local.dir: "/tmp/storm-data"
```

We are installing two master nodes.

5. Go to the `$STORM_HOME/conf` directory at each worker node and add the following lines to the `storm.yaml` file:

```
storm.zookeeper.servers:
- "zoo1"
- "zoo2"
- "zoo3"
storm.zookeeper.port: 2181
nimbus.seeds: "nimbus1,nimbus2"
storm.local.dir: "/tmp/storm-data"
supervisor.slots.ports:
- 6700
- 6701
- 6702
- 6703
```

If you are planning to execute the Nimbus and supervisor on the same machine, then add the `supervisor.slots.ports` property to the Nimbus machine too.

6. Go to the `$STORM_HOME` directory at the master nodes and execute the following command to start the master daemon:

   ```
   $> bin/storm nimbus &
   ```

7. Go to the `$STORM_HOME` directory at each worker node (or supervisor node) and execute the following command to start the worker daemons:

   ```
   $> bin/storm supervisor &
   ```

Developing the hello world example

Before starting the development, you should have Eclipse and Maven installed in your project. The sample topology explained here will cover how to create a basic Storm project, including a spout and bolt, and how to build, and execute them.

Create a Maven project by using `com.stormadvance` as `groupId` and `storm-example` as `artifactId`.

Add the following Maven dependencies to the `pom.xml` file:

```
<dependency>
  <groupId>org.apache.storm</groupId>
  <artifactId>storm-core</artifactId>
  <version>1.0.2</version>
  <scope>provided<scope>
</dependency>
```

TIP

Make sure the scope of the Storm dependency is provided, otherwise you will not be able to deploy the topology on the Storm cluster.

Add the following Maven `build` plugins in the `pom.xml` file:

```
<build>
  <plugins>
    <plugin>
      <artifactId>maven-assembly-plugin</artifactId>
      <version>2.2.1</version>
      <configuration>
        <descriptorRefs>
          <descriptorRef>jar-with-dependencies
          </descriptorRef>
```

```
          </descriptorRefs>
          <archive>
            <manifest>
              <mainClass />
            </manifest>
          </archive>
        </configuration>
        <executions>
          <execution>
            <id>make-assembly</id>
            <phase>package</phase>
            <goals>
              <goal>single</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
```

Write your first sample spout by creating a `SampleSpout` class in the `com.stormadvance.storm_example` package. The `SampleSpout` class extends the serialized `BaseRichSpout` class. This spout does not connect to an external source to fetch data, but randomly generates the data and emits a continuous stream of records. The following is the source code of the `SampleSpout` class with an explanation:

```
public class SampleSpout extends BaseRichSpout {
  private static final long serialVersionUID = 1L;

  private static final Map<Integer, String> map = new HashMap<Integer,
String>();
  static {
    map.put(0, "google");
    map.put(1, "facebook");
    map.put(2, "twitter");
    map.put(3, "youtube");
    map.put(4, "linkedin");
  }
  private SpoutOutputCollector spoutOutputCollector;

  public void open(Map conf, TopologyContext context, SpoutOutputCollector
spoutOutputCollector) {
    // Open the spout
    this.spoutOutputCollector = spoutOutputCollector;
  }

  public void nextTuple() {
      // Storm cluster repeatedly calls this method to emita continuous
      // stream of tuples.
      final Random rand = new Random();
      // generate the random number from 0 to 4.
      int randomNumber = rand.nextInt(5);
      spoutOutputCollector.emit(new Values(map.get(randomNumber)));
      try{
        Thread.sleep(5000);
      }catch(Exception e) {
        System.out.println("Failed to sleep the thread");
      }
    }

    public void declareOutputFields(OutputFieldsDeclarer declarer) {

    // emit the tuple with field "site"
    declarer.declare(new Fields("site"));
    }
  }
```

Write your first sample bolt by creating a `SampleBolt` class within the same package. The `SampleBolt` class extends the serialized `BaseRichBolt` class. This bolt will consume the tuples emitted by the `SampleSpout` spout and will print the value of the field `site` on the console. The following is the source code of the `SampleStormBolt` class with an explanation:

```
  public class SampleBolt extends BaseBasicBolt {
    private static final long serialVersionUID = 1L;

    public void execute(Tuple input, BasicOutputCollector collector) {
      // fetched the field "site" from input tuple.
      String test = input.getStringByField("site");
      // print the value of field "site" on console.
      System.out.println("######### Name of input site is : " + test);
    }

    public void declareOutputFields(OutputFieldsDeclarer declarer) {
    }
  }
```

Create a main `SampleStormTopology` class within the same package. This class creates an instance of the spout and bolt along with the classes, and chaines them together using a `TopologyBuilder` class. This class uses `org.apache.storm.LocalCluster` to simulate the Storm cluster. The `LocalCluster` mode is used for debugging/testing the topology on a developer machine before deploying it on the Storm cluster. The following is the implementation of the main class:

```
public class SampleStormTopology {
  public static void main(String[] args) throws AlreadyAliveException,
InvalidTopologyException {
    // create an instance of TopologyBuilder class
    TopologyBuilder builder = new TopologyBuilder();
    // set the spout class
    builder.setSpout("SampleSpout", new SampleSpout(), 2);
    // set the bolt class
    builder.setBolt("SampleBolt", new SampleBolt(),
4).shuffleGrouping("SampleSpout");
    Config conf = new Config();
    conf.setDebug(true);
    // create an instance of LocalCluster class for
    // executing topology in local mode.
    LocalCluster cluster = new LocalCluster();
    // SampleStormTopology is the name of submitted topology
    cluster.submitTopology("SampleStormTopology", conf,
builder.createTopology());
    try {
      Thread.sleep(100000);
    } catch (Exception exception) {
      System.out.println("Thread interrupted exception : " + exception);
    }
    // kill the SampleStormTopology
    cluster.killTopology("SampleStormTopology");
    // shutdown the storm test cluster
    cluster.shutdown();
  }
}
```

Go to your project's home directory and run the following commands to execute the topology in local mode:

```
$> cd $STORM_EXAMPLE_HOME
$> mvn compile exec:java -Dexec.classpathScope=compile -
Dexec.mainClass=com.stormadvance.storm_example.SampleStormTopology
```

Now create a new topology class for deploying the topology on an actual Storm cluster. Create a main `SampleStormClusterTopology` class within the same package. This class also creates an instance of the spout and bolt along with the classes, and chains them together using a `TopologyBuilder` class:

```
public class SampleStormClusterTopology {
  public static void main(String[] args) throws AlreadyAliveException,
InvalidTopologyException {
    // create an instance of TopologyBuilder class
    TopologyBuilder builder = new TopologyBuilder();
    // set the spout class
    builder.setSpout("SampleSpout", new SampleSpout(), 2);
    // set the bolt class
    builder.setBolt("SampleBolt", new SampleBolt(),
4).shuffleGrouping("SampleSpout");
    Config conf = new Config();
    conf.setNumWorkers(3);
    // This statement submit the topology on remote
    // args[0] = name of topology
    try {
      StormSubmitter.submitTopology(args[0], conf,
builder.createTopology());
    } catch (AlreadyAliveException alreadyAliveException) {
      System.out.println(alreadyAliveException);
    } catch (InvalidTopologyException invalidTopologyException) {
      System.out.println(invalidTopologyException);
    } catch (AuthorizationException e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
    }
  }
}
```

Build your Maven project by running the following command on the projects home directory:

```
mvn clean install
```

The output of the preceding command is as follows:

```
    ------------------------------------------------------------------ ----
-
    [INFO] ----------------------------------------------------------- ----
-
    [INFO] BUILD SUCCESS
    [INFO] ----------------------------------------------------------- ----
-
    [INFO] Total time: 58.326s
    [INFO] Finished at:
    [INFO] Final Memory: 14M/116M
    [INFO] ------------------------------------------------------------ ----
```

We can deploy the topology to the cluster using the following Storm client command:

```
bin/storm jar jarName.jar [TopologyMainClass] [Args]
```

The preceding command runs `TopologyMainClass` with the arguments `arg1` and `arg2`. The main function of `TopologyMainClass` is to define the topology and submit it to the Nimbus machine. The `storm jar` part takes care of connecting to the Nimbus machine and uploading the JAR part.

Log in on a Storm Nimbus machine and execute the following commands:

```
$> cd $STORM_HOME
$> bin/storm jar ~/storm_example-0.0.1-SNAPSHOT-jar-with-dependencies.jar
com.stormadvance.storm_example.SampleStormClusterTopology storm_example
```

In the preceding code `~/storm_example-0.0.1-SNAPSHOT-jar-with-dependencies.jar` is the path of the `SampleStormClusterTopology` JAR that we are deploying on the Storm cluster.

The following information is displayed:

```
702  [main] INFO  o.a.s.StormSubmitter - Generated ZooKeeper secret payload
for MD5-digest: -8367952358273199959:-5050558042400210383
793  [main] INFO  o.a.s.s.a.AuthUtils - Got AutoCreds []
856  [main] INFO  o.a.s.StormSubmitter - Uploading topology jar
/home/USER/storm_example-0.0.1-SNAPSHOT-jar-with-dependencies.jar to
assigned location: /tmp/storm-data/nimbus/inbox/stormjar-d3007821-
f87d-48af-8364-cff7abf8652d.jar
867  [main] INFO  o.a.s.StormSubmitter - Successfully uploaded topology jar
to assigned location: /tmp/storm-data/nimbus/inbox/stormjar-d3007821-
f87d-48af-8364-cff7abf8652d.jar
868  [main] INFO  o.a.s.StormSubmitter - Submitting topology storm_example
in distributed mode with conf
{"storm.zookeeper.topology.auth.scheme":"digest","storm.zookeeper.topology.
auth.payload":"-8367952358273199959:-5050558042400210383","topology.workers
":3}
 1007 [main] INFO  o.a.s.StormSubmitter - Finished submitting topology:
storm_example
```

Run the `jps` command to see the number of running JVM processes as follows:

```
jps
```

The preceding command's output is:

```
26827 worker
26530 supervisor
26824 worker
26468 nimbus
26822 worker
```

In the preceding code, a `worker` is the JVM launched for the `SampleStormClusterTopology` topology.

The different options of the Storm topology

This section covers the following operations that a user can perform on the Storm cluster:

- Deactivate
- Activate
- Rebalance
- Kill
- Dynamic log level settings

Deactivate

Storm supports the deactivating a topology. In the deactivated state, spouts will not emit any new tuples into the pipeline, but the processing of the already emitted tuples will continue. The following is the command to deactivate the running topology:

```
$> bin/storm deactivate topologyName
```

Deactivate `SampleStormClusterTopology` using the following command:

```
bin/storm deactivate SampleStormClusterTopology
```

The following information is displayed:

```
0 [main] INFO backtype.storm.thrift - Connecting to Nimbus at
localhost:6627
76 [main] INFO backtype.storm.command.deactivate - Deactivated topology:
SampleStormClusterTopology
```

Activate

Storm also the supports activating a topology. When a topology is activated, spouts will again start emitting tuples. The following is the command to activate the topology:

```
$> bin/storm activate topologyName
```

Activate `SampleStormClusterTopology` using the following command:

```
bin/storm activate SampleStormClusterTopology
```

The following information is displayed:

```
0 [main] INFO backtype.storm.thrift - Connecting to Nimbus at
localhost:6627
65 [main] INFO backtype.storm.command.activate - Activated topology:
SampleStormClusterTopology
```

Rebalance

The process of updating a the topology parallelism at the runtime is called a **rebalance**. A more detailed information of this operation acn be in `Chapter 3`, *Storm Parallelism and Data Partitioning*.

Kill

Storm topologies are never-ending processes. To stop a topology, we need to kill it. When killed, the topology first enters into the deactivation state, processes all the tuples already emitted into it, and then stops. Run the following command to kill `SampleStormClusterTopology`:

```
$> bin/storm kill SampleStormClusterTopology
```

The following information is displayed:

```
0 [main] INFO backtype.storm.thrift - Connecting to Nimbus at
localhost:6627
80 [main] INFO backtype.storm.command.kill-topology - Killed topology:
SampleStormClusterTopology
```

Now, run the `jps` command again to see the remaining JVM processes as follows:

```
jps
```

The preceding command's output is:

```
26530 supervisor
27193 Jps
26468 nimbus
```

Dynamic log level settings

This allows the user to change the log level of topology on runtime without stopping the topology. The detailed information of this operation can be found at the end of this chapter.

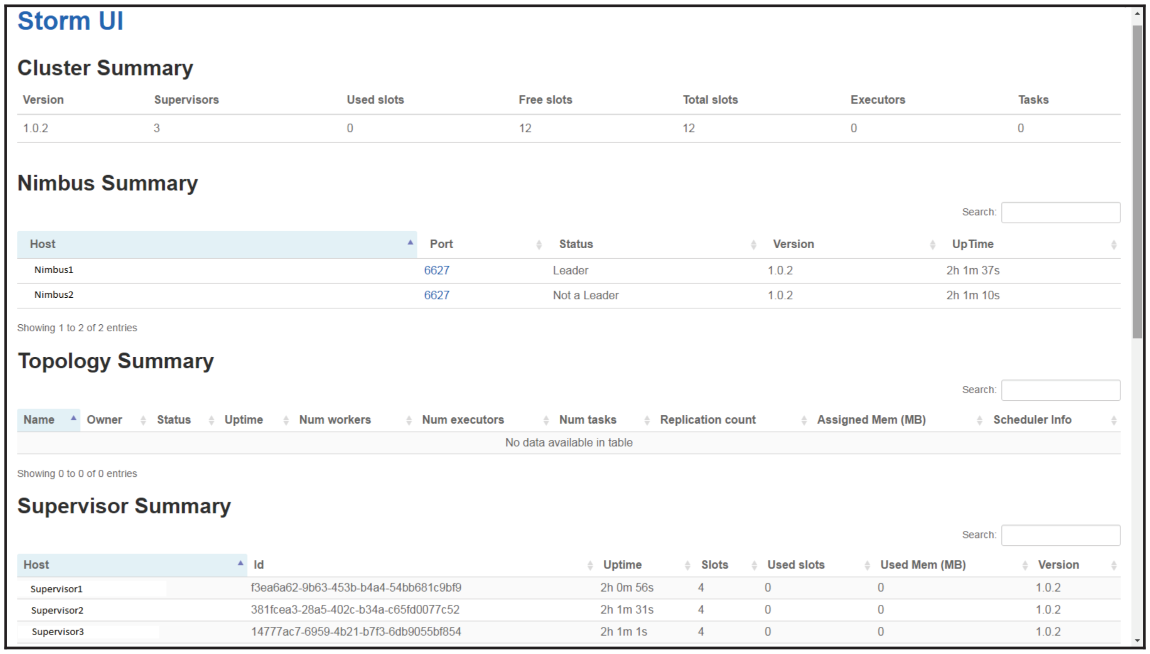
Walkthrough of the Storm UI

This section will show you how we can start the Storm UI daemon. However, before starting the Storm UI daemon, we assume that you have a running Storm cluster. The Storm cluster deployment steps are mentioned in the previous sections of this chapter. Now, go to the Storm home directory (`cd $STORM_HOME`) at the leader Nimbus machine and run the following command to start the Storm UI daemon:

```
$> cd $STORM_HOME
$> bin/storm ui &
```

By default, the Storm UI starts on the `8080` port of the machine where it is started. Now, we will browse to the `http://nimbus-node:8080` page to view the Storm UI, where Nimbus node is the IP address or hostname of the the Nimbus machine.

The following is a screenshot of the Storm home page:



Storm UI

Cluster Summary

Version	Supervisors	Used slots	Free slots	Total slots	Executors	Tasks
1.0.2	3	0	12	12	0	0

Nimbus Summary

Search:

Host	Port	Status	Version	UpTime
Nimbus1	6627	Leader	1.0.2	2h 1m 37s
Nimbus2	6627	Not a Leader	1.0.2	2h 1m 10s

Showing 1 to 2 of 2 entries

Topology Summary

Search:

Name	Owner	Status	Uptime	Num workers	Num executors	Num tasks	Replication count	Assigned Mem (MB)	Scheduler Info
No data available in table									

Showing 0 to 0 of 0 entries

Supervisor Summary

Search:

Host	Id	Uptime	Slots	Used slots	Used Mem (MB)	Version
Supervisor1	f3ea6a62-9b63-453b-b4a4-54bb681c9bf9	2h 0m 56s	4	0	0	1.0.2
Supervisor2	381fcea3-28a5-402c-b34a-c65fd0077c52	2h 1m 31s	4	0	0	1.0.2
Supervisor3	14777ac7-6959-4b21-b7f3-6db9055bf854	2h 1m 1s	4	0	0	1.0.2

Cluster Summary section

This portion of the Storm UI shows the version of Storm deployed in the cluster, the uptime of the Nimbus nodes, number of free worker slots, number of used worker slots, and so on. While submitting a topology to the cluster, the user first needs to make sure that the value of the **Free slots** column should not be zero; otherwise, the topology doesn't get any worker for processing and will wait in the queue until a workers becomes free.

Nimbus Summary section

This portion of the Storm UI shows the number of Nimbus processes that are running in a Storm cluster. The section also shows the status of the Nimbus nodes. A node with the status `Leader` is an active master while the node with the status `Not a Leader` is a passive master.

Supervisor Summary section

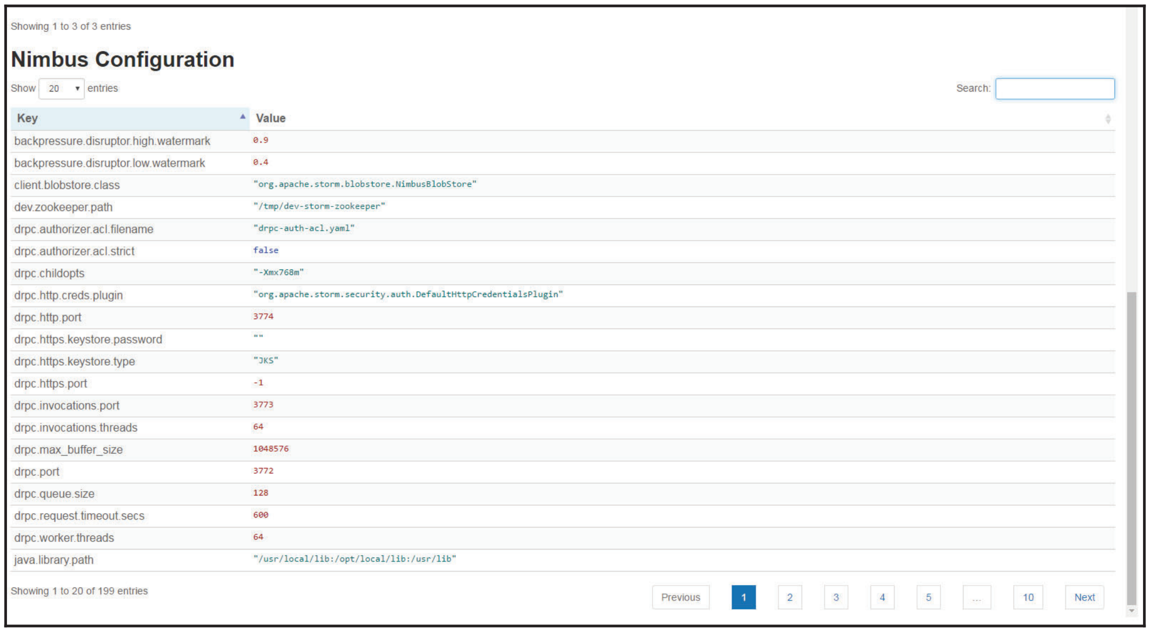
This portion of the Storm UI shows the list of supervisor nodes running in the cluster, along with their **Id**, **Host**, **Uptime**, **Slots**, and **Used slots** columns.

Nimbus Configuration section

This portion of the Storm UI shows the configuration of the Nimbus node. Some of the important properties are:

- `supervisor.slots.ports`
- `storm.zookeeper.port`
- `storm.zookeeper.servers`
- `storm.zookeeper.retry.interval`
- `worker.childopts`
- `supervisor.childopts`

The following is a screenshot of **Nimbus Configuration**:



Nimbus Configuration

Key	Value
backpressure.disruptor.high.watermark	0.9
backpressure.disruptor.low.watermark	0.4
client.blobstore.class	"org.apache.storm.blobstore.NimbusBlobStore"
dev.zookeeper.path	"/tmp/dev-storm-zookeeper"
drpc.authorizer.acl.filename	"drpc-auth-acl.yaml"
drpc.authorizer.acl.strict	false
drpc.childopts	"-Xmx768m"
drpc.http.creds.plugin	"org.apache.storm.security.auth.DefaultHttpCredentialsPlugin"
drpc.http.port	3774
drpc.https.keystore.password	""
drpc.https.keystore.type	"JKS"
drpc.https.port	-1
drpc.invocations.port	3773
drpc.invocations.threads	64
drpc.max_buffer_size	1048576
drpc.port	3772
drpc.queue.size	128
drpc.request.timeout.secs	600
drpc.worker.threads	64
java.library.path	"/usr/local/lib:/opt/local/lib:/usr/lib"

Topology Summary section

This portion of the Storm UI shows the list of topologies running in the Storm cluster, along with their ID, the number of workers assigned to the topology, the number of executors, number of tasks, uptime, and so on.

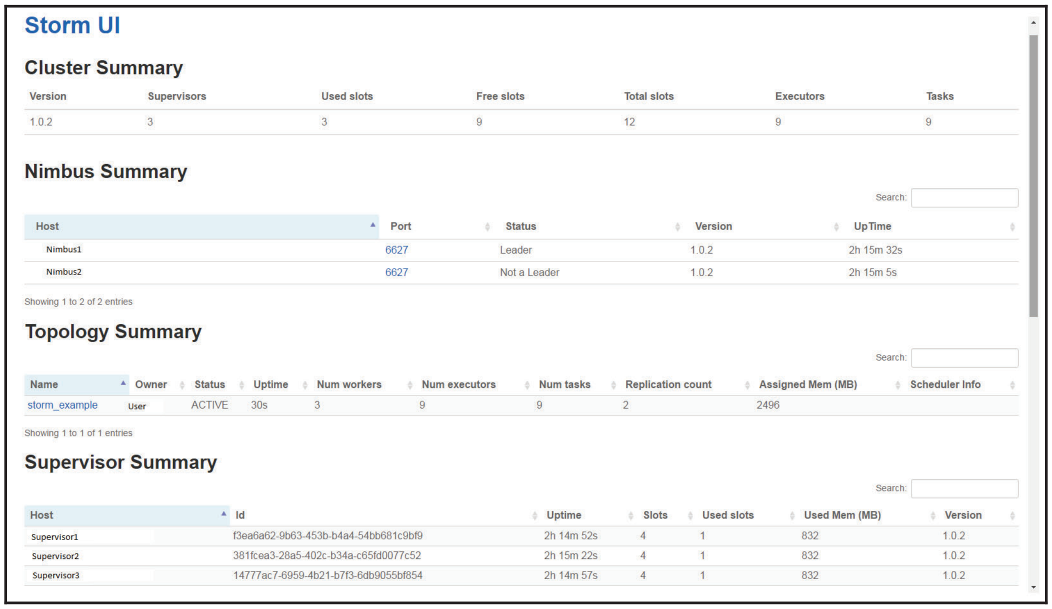
Let's deploy the sample topology (if it is not running already) in a remote Storm cluster by running the following command:

```
$> cd $STORM_HOME
$> bin/storm jar ~/storm_example-0.0.1-SNAPSHOT-jar-with-dependencies.jar
com.stormadvance.storm_example.SampleStormClusterTopology storm_example
```

We have created the `SampleStormClusterTopology` topology by defining three worker processes, two executors for `SampleSpout`, and four executors for `SampleBolt`.

After submitting `SampleStormClusterTopology` on the Storm cluster, the user has to refresh the Storm home page.

The following screenshot shows that the row is added for `SampleStormClusterTopology` in the **Topology Summary** section. The topology section contains the name of the topology, unique ID of the topology, status of the topology, uptime, number of workers assigned to the topology, and so on. The possible values of the **Status** fields are `ACTIVE`, `KILLED`, and `INACTIVE`.



Storm UI

Cluster Summary

Version	Supervisors	Used slots	Free slots	Total slots	Executors	Tasks
1.0.2	3	3	9	12	9	9

Nimbus Summary

Search:

Host	Port	Status	Version	UpTime
Nimbus1	6627	Leader	1.0.2	2h 15m 32s
Nimbus2	6627	Not a Leader	1.0.2	2h 15m 5s

Showing 1 to 2 of 2 entries

Topology Summary

Search:

Name	Owner	Status	Uptime	Num workers	Num executors	Num tasks	Replication count	Assigned Mem (MB)	Scheduler Info
storm_example	User	ACTIVE	30s	3	9	9	2	2496	

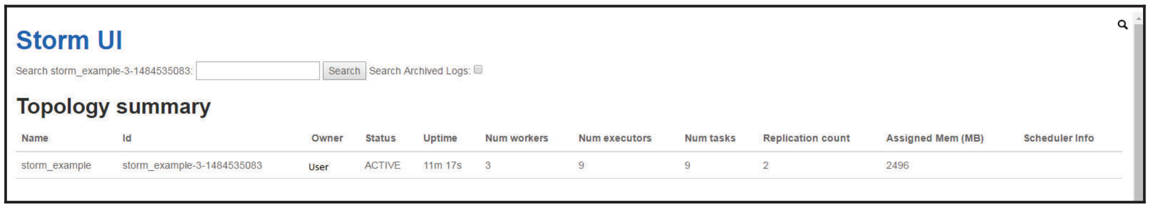
Showing 1 to 1 of 1 entries

Supervisor Summary

Search:

Host	Id	Uptime	Slots	Used slots	Used Mem (MB)	Version
Supervisor1	f3ea6a62-9b63-453b-b4a4-54bb681c9bf9	2h 14m 52s	4	1	832	1.0.2
Supervisor2	381fcea3-28a5-402c-b34a-c65fd0077c52	2h 15m 22s	4	1	832	1.0.2
Supervisor3	14777ac7-6959-4b21-b7f3-6db9055bf854	2h 14m 57s	4	1	832	1.0.2

Let's click on `SampleStormClusterTopology` to view its detailed statistics. There are two screenshots for this. The first one contains the information about the number of workers, executors, and tasks assigned to the `SampleStormClusterTopology` topology:



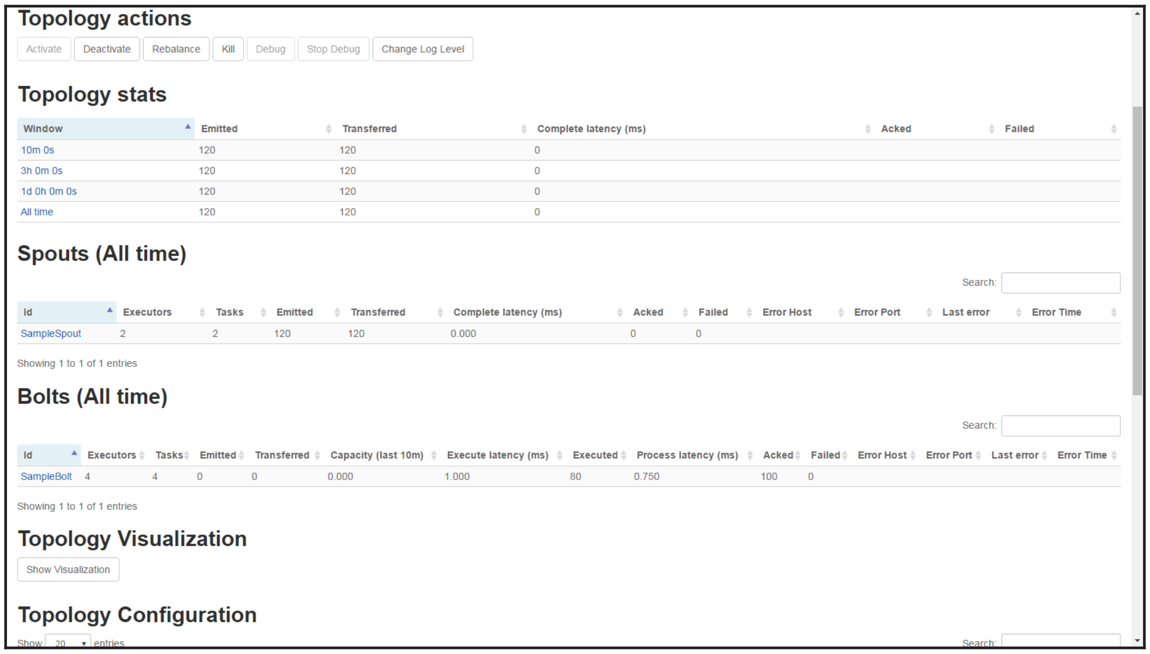
Storm UI

Search storm_example-3-1484535083: Search Search Archived Logs:

Topology summary

Name	Id	Owner	Status	Uptime	Num workers	Num executors	Num tasks	Replication count	Assigned Mem (MB)	Scheduler Info
storm_example	storm_example-3-1484535083	User	ACTIVE	11m 17s	3	9	9	2	2496	

The next screenshot contains information about the spouts and bolts, including the number of executors and tasks assigned to each spout and bolt:



Topology actions

Activate Deactivate Rebalance Kill Debug Stop Debug Change Log Level

Topology stats

Window	Emitted	Transferred	Complete latency (ms)	Acked	Failed
10m 0s	120	120	0		
3h 0m 0s	120	120	0		
1d 0h 0m 0s	120	120	0		
All time	120	120	0		

Spouts (All time)

Search:

Id	Executors	Tasks	Emitted	Transferred	Complete latency (ms)	Acked	Failed	Error Host	Error Port	Last error	Error Time
SampleSpout	2	2	120	120	0.000	0	0				

Showing 1 to 1 of 1 entries

Bolts (All time)

Search:

Id	Executors	Tasks	Emitted	Transferred	Capacity (last 10m)	Execute latency (ms)	Executed	Process latency (ms)	Acked	Failed	Error Host	Error Port	Last error	Error Time
SampleBolt	4	4	0	0	0.000	1.000	80	0.750	100	0				

Showing 1 to 1 of 1 entries

Topology Visualization

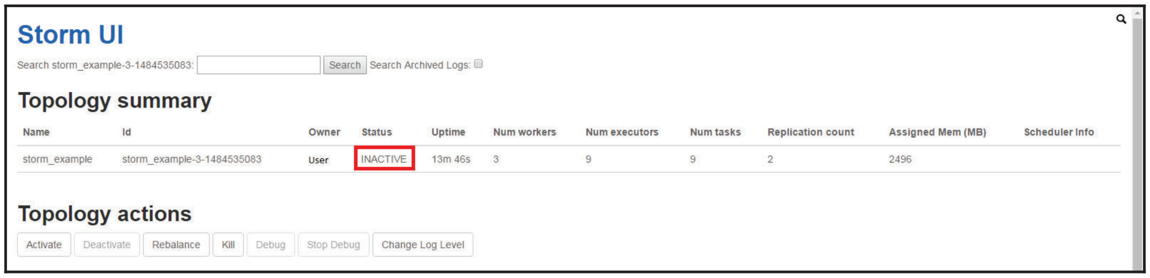
Show Visualization

Topology Configuration

Show 20 entries Search:

The information shown in the previous screenshots is as follows:

- **Topology stats**: This section will give information about the number of tuples emitted, transferred, and acknowledged, the capacity latency, and so on, within the windows of 10 minutes, 3 hours, 1 day, and since the start of the topology
- **Spouts (All time)**: This section shows the statistics of all the spouts running inside the topology
- **Bolts (All time)**: This section shows the statistics of all the bolts running inside the topology
- **Topology actions**: This section allows us to perform activate, deactivate, rebalance, kill, and other operations on the topologies directly through the Storm UI:
    - **Deactivate**: Click on **Deactivate** to deactivate the topology. Once the topology is deactivated, the spout stops emitting tuples and the status of the topology changes to **INACTIVE** on the Storm UI.



Storm UI

Search storm_example-3-1484535083: Search Search Archived Logs:

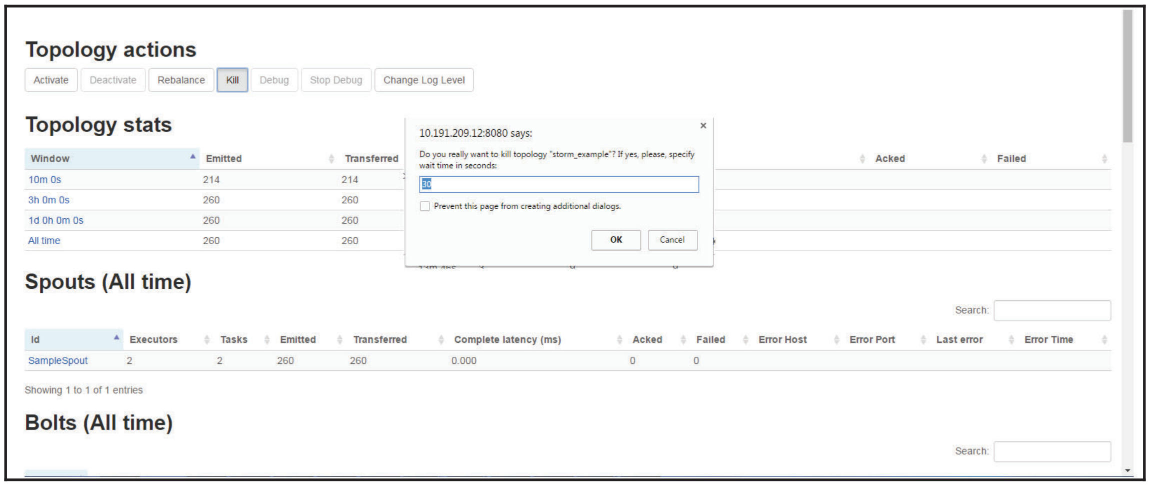
Topology summary

Name	Id	Owner	Status	Uptime	Num workers	Num executors	Num tasks	Replication count	Assigned Mem (MB)	Scheduler Info
storm_example	storm_example-3-1484535083	User	INACTIVE	13m 46s	3	9	9	2	2496	

Topology actions

Deactivating the topology does not free the Storm resource.

- **Activate**: Click on the **Activate** button to activate the topology. Once the topology is activated, the spout again starts emitting tuples.
- **Kill**: Click on the **Kill** button to destroy/kill the topology. Once the topology is killed, it will free all the Storm resources allotted to this topology. While killing the topology, the Storm will first deactivate the spouts and wait for the kill time mentioned on the alerts box so that the bolts have a chance to finish the processing of the tuples emitted by the spouts before the kill command. The following screenshot shows how we can kill the topology through the Storm UI:



Topology actions

Activate | Deactivate | Rebalance | Kill | Debug | Stop Debug | Change Log Level

Topology stats

Window	Emitted	Transferred	Acked	Failed
10m 0s	214	214		
3h 0m 0s	260	260		
1d 0h 0m 0s	260	260		
All time	260	260		

10.191.209.12:8080 says:

Do you really want to kill topology "storm_example"? If yes, please, specify wait time in seconds:

30

Prevent this page from creating additional dialogs.

OK Cancel

Spouts (All time)

Search:

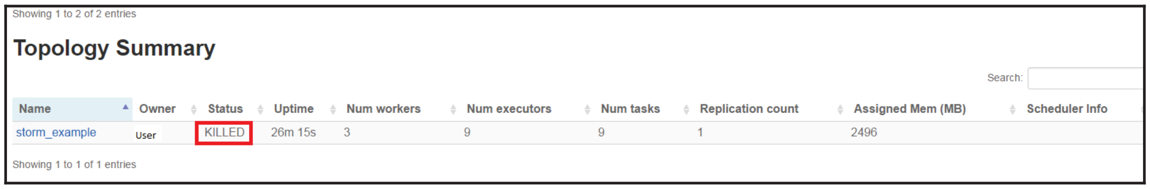
Id	Executors	Tasks	Emitted	Transferred	Complete latency (ms)	Acked	Failed	Error Host	Error Port	Last error	Error Time
SampleSpout	2	2	260	260	0.000	0	0				

Showing 1 to 1 of 1 entries

Bolts (All time)

Search:

Let's go to the Storm UI's home page to check the status of `SampleStormClusterToplogy`, as shown in the following screenshot:



Showing 1 to 2 of 2 entries

Topology Summary

Search:

Name	Owner	Status	Uptime	Num workers	Num executors	Num tasks	Replication count	Assigned Mem (MB)	Scheduler Info
storm_example	User	KILLED	26m 15s	3	9	9	1	2496	

Showing 1 to 1 of 1 entries

Dynamic log level settings

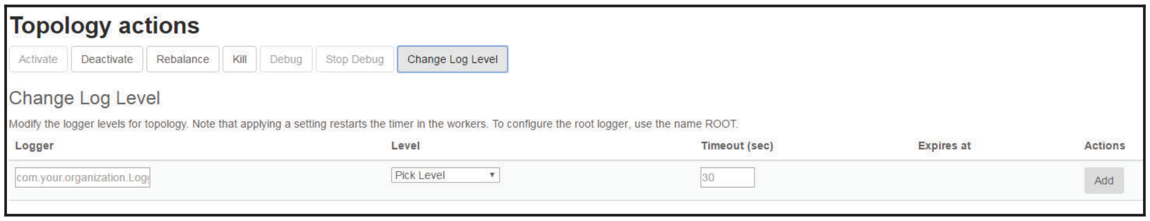
The dynamic log level allows us to change the log level setting of the topology on the runtime from the Storm CLI and the Storm UI.

Updating the log level from the Storm UI

Go through the following steps to update the log level from the Storm UI:

1. Deploy `SampleStormClusterTopology` again on the Storm cluster if it is not running.
2. Browse the Storm UI at `http://nimbus-node:8080/`.
3. Click on the `storm_example` topology.

4. Now click on the **Change Log Level** button to change the `ROOT` logger of the topology, as shown in the following are the screenshots:



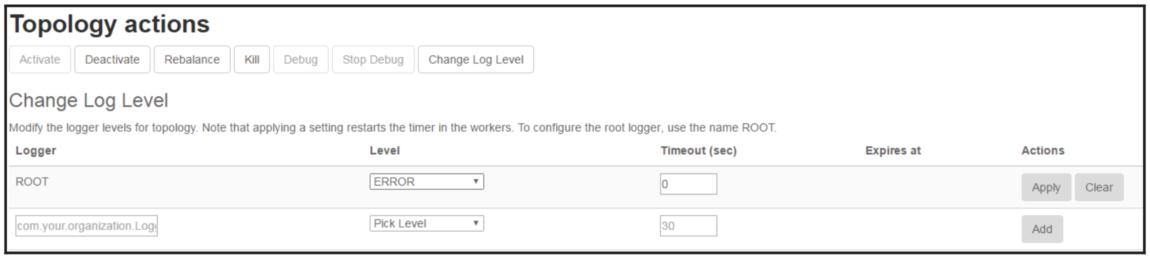
Topology actions

Change Log Level

Modify the logger levels for topology. Note that applying a setting restarts the timer in the workers. To configure the root logger, use the name ROOT.

Logger	Level	Timeout (sec)	Expires at	Actions
com.your.organization.Log	Pick Level	30		Add

5. Configure the entries mentioned in the following screenshots change the `ROOT` logger to **ERROR**:



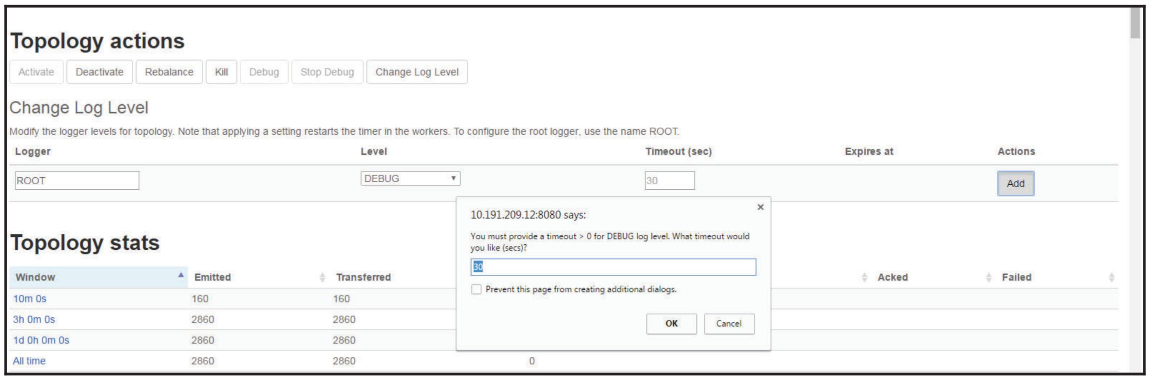
Topology actions

Change Log Level

Modify the logger levels for topology. Note that applying a setting restarts the timer in the workers. To configure the root logger, use the name ROOT.

Logger	Level	Timeout (sec)	Expires at	Actions
ROOT	ERROR	0		Apply Clear
com.your.organization.Log	Pick Level	30		Add

6. If you are planning to change the logging level to **DEBUG**, then you must specify the timeout (expiry time) for that log level, as shown in the following screenshots:



Topology actions

Change Log Level

Modify the logger levels for topology. Note that applying a setting restarts the timer in the workers. To configure the root logger, use the name ROOT.

Logger	Level	Timeout (sec)	Expires at	Actions
ROOT	DEBUG	30		Add

10.191.209.12:8080 says:

You must provide a timeout > 0 for DEBUG log level. What timeout would you like (secs)?

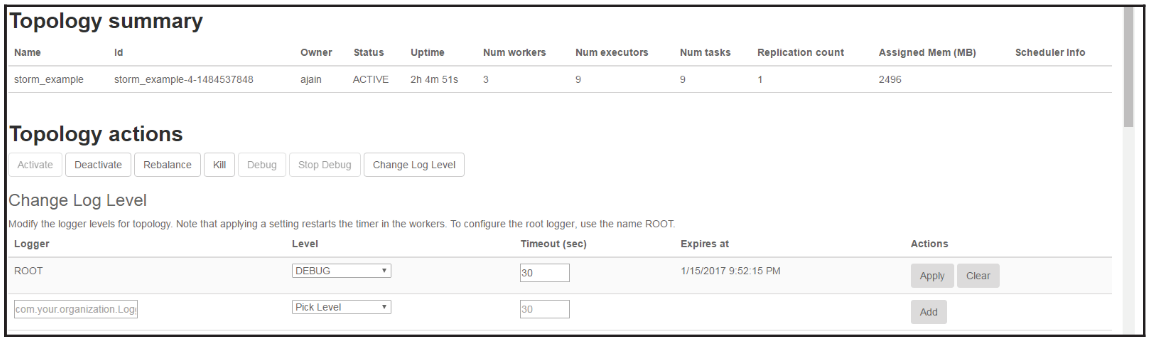
30

Prevent this page from creating additional dialogs.

Topology stats

Window	Emitted	Transferred		Acked	Failed
10m 0s	160	160			
3h 0m 0s	2860	2860			
1d 0h 0m 0s	2860	2860			
All time	2860	2860	0		

7. Once the time mentioned in the expiry time is reached, the log level will go back to the default value:



Topology summary

Name	Id	Owner	Status	Uptime	Num workers	Num executors	Num tasks	Replication count	Assigned Mem (MB)	Scheduler Info
storm_example	storm_example-4-1484537848	ajain	ACTIVE	2h 4m 51s	3	9	9	1	2496	

Topology actions

Change Log Level

Modify the logger levels for topology. Note that applying a setting restarts the timer in the workers. To configure the root logger, use the name ROOT.

Logger	Level	Timeout (sec)	Expires at	Actions
ROOT	DEBUG	30	1/15/2017 9:52:15 PM	Apply Clear
com.your.organization.Log	Pick Level	30		Add

8. **Clear** button mentioned in the **Action** column will clear the log setting, and the application will set the default log setting again.

Updating the log level from the Storm CLI

We can modify the log level from the Storm CLI. The following is the command that the user has to execute from the Storm directory to update the log settings on the runtime:

```
bin/storm set_log_level [topology name] -l [logger name]=[LEVEL]:[TIMEOUT]
```

In the preceding code, `topology name` is the name of the topology, and `logger name` is the logger we want to change. If you want to change the `ROOT` logger, then use `ROOT` as a value of `logger name`. The `LEVEL` is the log level you want to apply. The possible values are `DEBUG`, `INFO`, `ERROR`, `TRACE`, `ALL`, `WARN`, `FATAL`, and `OFF`.

The `TIMEOUT` is the time in seconds. The log level will go back to normal after the timeout time. The value of `TIMEOUT` is mandatory if you are setting the log level to `DEBUG`/`ALL`.

The following is the command to change the log level setting for the `storm_example` topology:

```
$> bin/storm set_log_level storm_example -l ROOT=DEBUG:30
```

The following is the command to clear the log level setting:

```
$> ./bin/storm set_log_level storm_example -r ROOT
```

Summary

In this chapter, we have covered the installation of Storm and ZooKeeper clusters, the deployment of topologies on Storm clusters, the high availability of Nimbus nodes, and topology monitoring through the Storm UI. We have also covered the different operations a user can perform on running topology. Finally, we focused on how we can change the log level of running topology.

In the next chapter, we will focus on the distribution of topologies on multiple Storm machines/nodes.

3
Storm Parallelism and Data Partitioning

In the first two chapters, we have covered the introduction to Storm, the installation of Storm, and developing a sample topology. In this chapter, we are focusing on distribution of the topology on multiple Storm machines/nodes. This chapter covers the following points:

- Parallelism of topology
- How to configure parallelism at the code level
- Different types of stream groupings in a Storm cluster
- Guaranteed message processing
- Tick tuple

Parallelism of a topology

Parallelism means the distribution of jobs on multiple nodes/instances where each instance can work independently and can contribute to the processing of data. Let's first look at the processes/components that are responsible for the parallelism of a Storm cluster.

Worker process

A Storm topology is executed across multiple supervisor nodes in the Storm cluster. Each of the nodes in the cluster can run one or more JVMs called **worker processes**, which are responsible for processing a part of the topology.

A worker process is specific to one of the specific topologies and can execute multiple components of that topology. If multiple topologies are being run at the same time, none of them will share any of the workers, thus providing some degree of isolation between topologies.

Executor

Within each worker process, there can be multiple threads executing parts of the topology. Each of these threads is called an **executor**. An executor can execute only one of the components, that is, any spout or bolt in the topology.

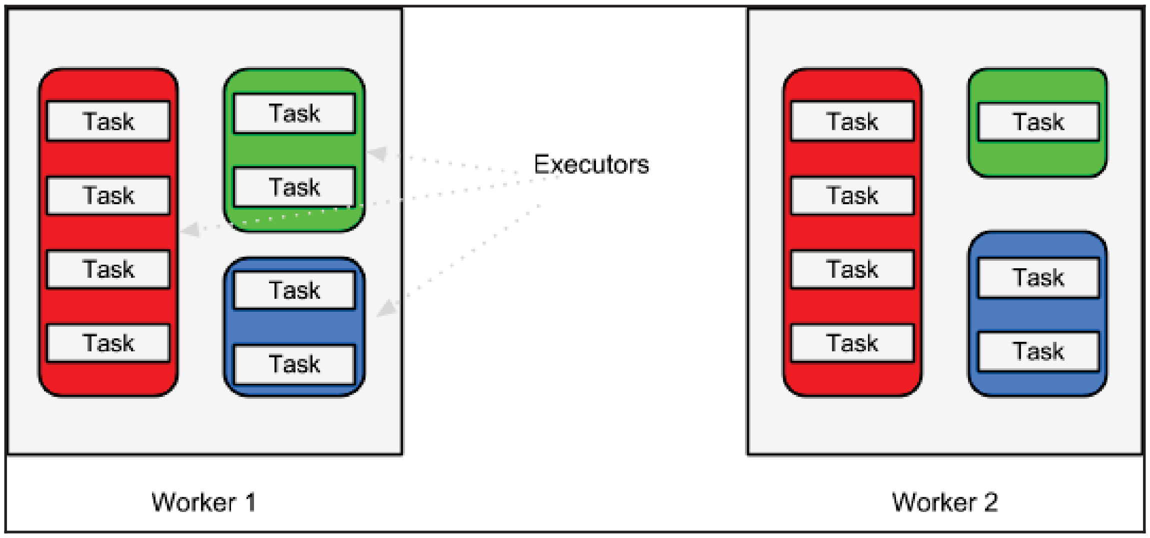
Each executor, being a single thread, can execute only tasks assigned to it serially. The number of executors defined for a spout or bolt can be changed dynamically while the topology is running, which means that you can easily control the degree of parallelism of various components in your topology.

Task

This is the most granular unit of task execution in Storm. Each task is an instance of a spout or bolt. When defining a Storm topology, you can specify the number of tasks for each spout and bolt. Once defined, the number of tasks cannot be changed for a component at runtime. Each task can be executed alone or with another task of the same type, or another instance of the same spout or bolt.

The following diagram depicts the relationship between a worker process, executors, and tasks. In the following diagram, there are two executors for each component, with each hosting a different number of tasks.

Also, as you can see, there are two executors and eight tasks defined for one component (each executor is hosting four tasks). If you are not getting enough performance out of this configuration, you can easily change the number of executors for the component to four or eight to increase performance and the tasks will be uniformly distributed between all executors of that component. The following diagrams show the relationship between executor, task, and worker:



Configure parallelism at the code level

Storm provides an API to set the number of worker processes, number of executors, and number of tasks at the code level. The following section shows how we can configure parallelism at the code level.

We can set the number of worker processes at the code level by using the `setNumWorkers` method of the `org.apache.storm.Config` class. Here is the code snippet to show these settings in practice:

```
Config conf = new Config();
conf.setNumWorkers(3);
```

In the previous chapter, we configured the number of workers as three. Storm will assign the three workers for the `SampleStormTopology` and `SampleStormClusterTopology` topology.

We can set the number of executors at the code level by passing the `parallelism_hint` argument in the `setSpout(args,args,parallelism_hint)` or `setBolt(args,args,parallelism_hint)` methods of the `org.apache.storm.topology.TopologyBuilder` class. Here is the code snippet to show these settings in practice:

```
builder.setSpout("SampleSpout", new SampleSpout(), 2);
// set the bolt class
builder.setBolt("SampleBolt", new SampleBolt(),
4).shuffleGrouping("SampleSpout");
```

In the previous chapter, we set `parallelism_hint=2` for `SampleSpout` and `parallelism_hint=4` for `SampleBolt`. At the time of execution, Storm will assign two executors for `SampleSpout` and four executors for `SampleBolt`.

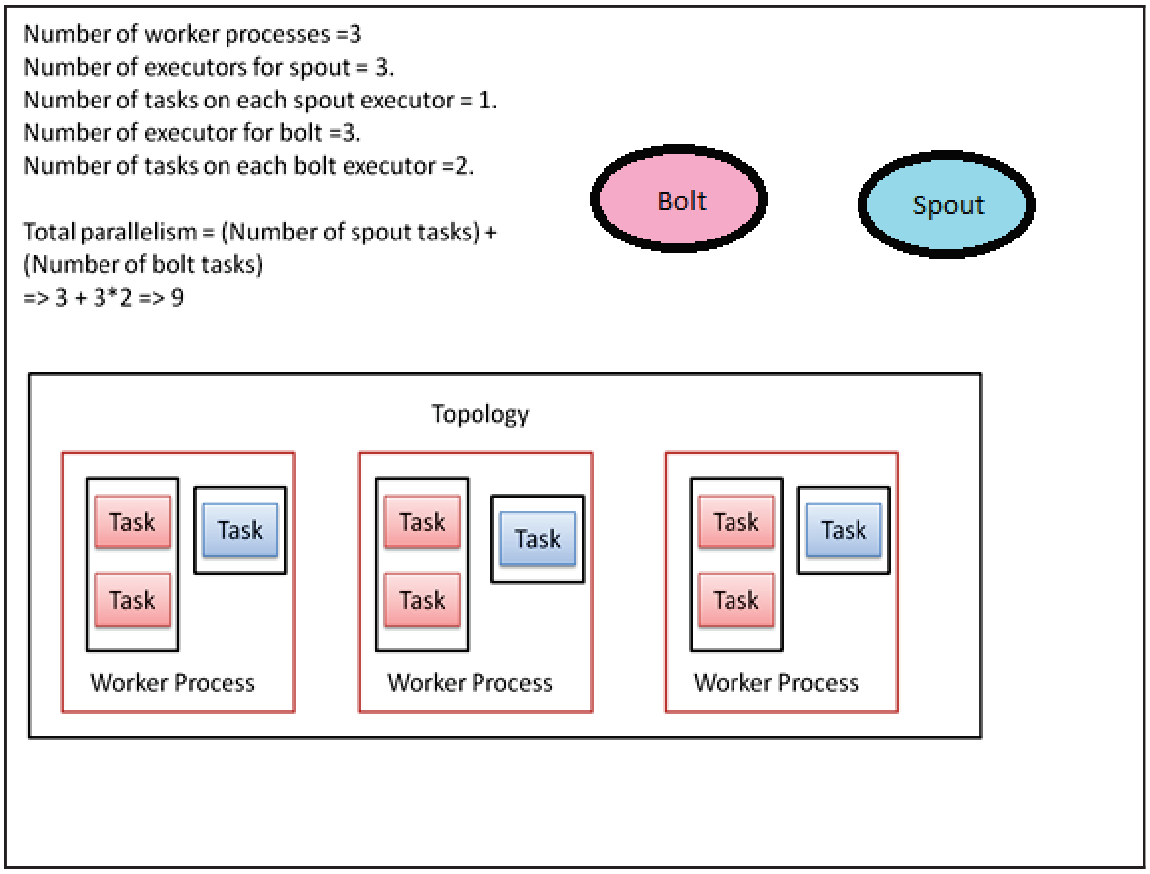
We can configure the number of tasks that can execute inside the executors. Here is the code snippet to show these settings in practice:

```
builder.setSpout("SampleSpout", new SampleSpout(), 2).setNumTasks(4);
```

In the preceding code, we have configured the two executors and four tasks of `SampleSpout`. For `SampleSpout`, Storm will assign two tasks per executor. By default, Storm will run one task per executor if the user does not set the number of tasks at the code level.

Worker process, executor, and task distribution

Let's assume the numbers of worker processes set for the topology is three, the number of executors for `SampleSpout` is three, and the number of executors for `SampleBolt` is three. Also, the number of tasks for `SampleBolt` is to be six, meaning that each `SampleBolt` executor will have two tasks. The following diagram shows what the topology would look like in operation:


Number of worker processes =3
Number of executors for spout = 3.
Number of tasks on each spout executor = 1.
Number of executor for bolt =3.
Number of tasks on each bolt executor =2.
Total parallelism = (Number of spout tasks) + (Number of bolt tasks)
=> 3 + 3*2 => 9
Bolt
Spout
Topology
Task
Task
Task
Worker Process
Task
Task
Task
Worker Process
Task
Task
Task
Worker Process

Rebalance the parallelism of a topology

As explained in the previous chapter, one of the key features of Storm is that it allows us to modify the parallelism of a topology at runtime. The process of updating a topology parallelism at runtime is called **rebalance**.

There are two ways to rebalance the topology:

- Using Storm Web UI
- Using Storm CLI

The Storm Web UI was covered in the previous chapter. This section covers how we can rebalance the topology using the Storm CLI tool. Here are the commands that we need to execute on Storm CLI to rebalance the topology:

```
> bin/storm rebalance [TopologyName] -n [NumberOfWorkers] -e
[Spout]=[NumberOfExecutos] -e [Bolt1]=[NumberOfExecutos]
[Bolt2]=[NumberOfExecutos]
```

The `rebalance` command will first deactivate the topology for the duration of the message timeout and then redistribute the workers evenly around the Storm cluster. After a few seconds or minutes, the topology will revert to the previous state of activation and restart the processing of input streams.

Rebalance the parallelism of a SampleStormClusterTopology topology

Let's first check the numbers of worker processes that are running in the Storm cluster by running the `jps` command on the supervisor machine:

Run the `jps` command on supervisor-1:

```
> jps
24347 worker
23940 supervisor
24593 Jps
24349 worker
```

Two worker processes are assigned to the supervisor-1 machine.

Now, run the `jps` command on supervisor-2:

```
> jps
24344 worker
23941 supervisor
24543 Jps
```

One worker process is assigned to the supervisor-2 machine.

A total of three worker processes are running on the Storm cluster.

Let's try reconfiguring `SampleStormClusterTopology` to use two worker processes, `SampleSpout` to use four executors, and `SampleBolt` to use four executors:

```
> bin/storm rebalance SampleStormClusterTopology -n 2 -e SampleSpout=4 -e
SampleBolt=4
0     [main] INFO  backtype.storm.thrift  - Connecting to Nimbus at
nimbus.host.ip:6627
58   [main] INFO  backtype.storm.command.rebalance  - Topology
SampleStormClusterTopology is rebalancing
```

Rerun the `jps` commands on the supervisor machines to view the number of worker processes.

Run the `jps` command on supervisor-1:

```
> jps
24377 worker
23940 supervisor
24593 Jps
```

Run the `jps` command on supervisor-2:

```
> jps
24353 worker
23941 supervisor
24543 Jps
```

In this case, two worker processes are shown previously. The first worker process is assigned to supervisor-1 and the other one is assigned to supervisor-2. The distribution of workers may vary depending on the number of topologies running on the system and the number of slots available on each supervisor. Ideally, Storm tries to distribute the load uniformly between all the nodes.

Different types of stream grouping in the Storm cluster

When defining a topology, we create a graph of computation with the number of bolt-processing streams. At a more granular level, each bolt executes multiple tasks in the topology. Thus, each task of a particular bolt will only get a subset of the tuples from the subscribed streams.

Stream grouping in Storm provides complete control over how this partitioning of tuples happens among the many tasks of a bolt subscribed to a stream. Grouping for a bolt can be defined on the instance of `org.apache.storm.topology.InputDeclarer` returned when defining bolts using the `org.apache e.storm.topology.TopologyBuilder.setBolt` method.

Storm supports the following types of stream groupings.

Shuffle grouping

Shuffle grouping distributes tuples in a uniform, random way across the tasks. An equal number of tuples will be processed by each task. This grouping is ideal when you want to distribute your processing load uniformly across the tasks and where there is no requirement for any data-driven partitioning. This is one of the most commonly used groupings in Storm.

Field grouping

This grouping enables you to partition a stream on the basis of some of the fields in the tuples. For example, if you want all the tweets from a particular user to go to a single task, then you can partition the tweet stream using field grouping by username in the following manner:

```
builder.setSpout("1", new TweetSpout());
builder.setBolt("2", new TweetCounter()).fieldsGrouping("1", new
Fields("username"))
```

As a result of the field grouping being *hash (fields) % (no. of tasks)*, it does not guarantee that each of the tasks will get tuples to process. For example, if you have applied a field grouping on a field, say *X*, with only two possible values, *A* and *B*, and created two tasks for the bolt, then it might be possible that both *hash (A) % 2* and *hash (B) % 2* return equal values, which will result in all the tuples being routed to a single task and the other being completely idle.

Another common usage of field grouping is to join streams. Since partitioning happens solely on the basis of field values, and not the stream type, we can join two streams with any common join fields. The name of the fields needs not be the same. For example, in the order processing domain, we can join the `Order` stream and the `ItemScanned` stream to see when an order is completed:

```
builder.setSpout("1", new OrderSpout());
builder.setSpount("2", new ItemScannedSpout());
builder.setBolt("joiner", new OrderJoiner())
.fieldsGrouping("1", new Fields("orderId"))
.fieldsGrouping("2", new Fields("orderRefId"));
```

Since joins on streams vary from application to application, you'll make your own definition of a join, say joins over a time window, that can be achieved by composing field groupings.

All grouping

All grouping is a special grouping that does not partition the tuples but replicates them to all the tasks, that is, each tuple will be sent to each of the bolt's tasks for processing.

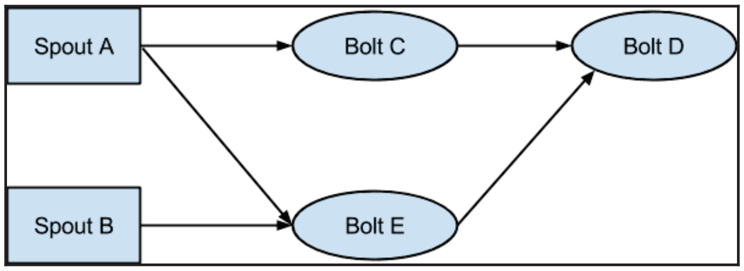
One common use case of all grouping is for sending signals to bolts. For example, if you are doing some kind of filtering on the streams, you can pass or change the filter parameters to all the bolts by sending them those parameters over a stream that is subscribed by all the bolt's tasks with an all grouping. Another example is to send a reset message to all the tasks in an aggregation bolt.

Global grouping

Global grouping does not partition the stream but sends the complete stream to the bolt's task, the smallest ID. A general use case of this is when there needs to be a reduce phase in your topology where you want to combine the results from previous steps in the topology into a single bolt.

Global grouping might seem redundant at first, as you can achieve the same results by defining the parallelism for the bolt as one if you only have one input stream. However, when you have multiple streams of data coming through a different path, you might want only one of the streams to be reduced and others to be parallel processes.

For example, consider the following topology. In this, you might want to combine all the tuples coming from **Bolt C** in a single **Bolt D** task, while you might still want parallelism for tuples coming from **Bolt E** to **Bolt D**:



Direct grouping

In direct grouping, the emitter decides where each tuple will go for processing. For example, say we have a log stream and we want to process each log entry to be processed by a specific bolt task on the basis of the type of resource. In this case, we can use direct grouping.

Direct grouping can only be used with direct streams. To declare a stream as a direct stream, use the `backtype.storm.topology.OutputFieldsDeclarer.declareStream` method, which takes a `boolean` parameter. Once you have a direct stream to emit to, use `backtype.storm.task.OutputCollector.emitDirect` instead of emit methods to emit it. The `emitDirect` method takes a `taskId` parameter to specify the task. You can get the number of tasks for a component using the `backtype.storm.task.TopologyContext.getComponentTasks` method.

Local or shuffle grouping

If the tuple source and target bolt tasks are running in the same worker, using this grouping will act as a shuffle grouping only between the target tasks running on the same worker, thus minimizing any network hops, resulting in increased performance.

If there are no target bolt tasks running on the source worker process, this grouping will act similar to the shuffle grouping mentioned earlier.

None grouping

None grouping is used when you don't care about the way tuples are partitioned among various tasks. As of Storm 0.8, this is equivalent to using shuffle grouping.

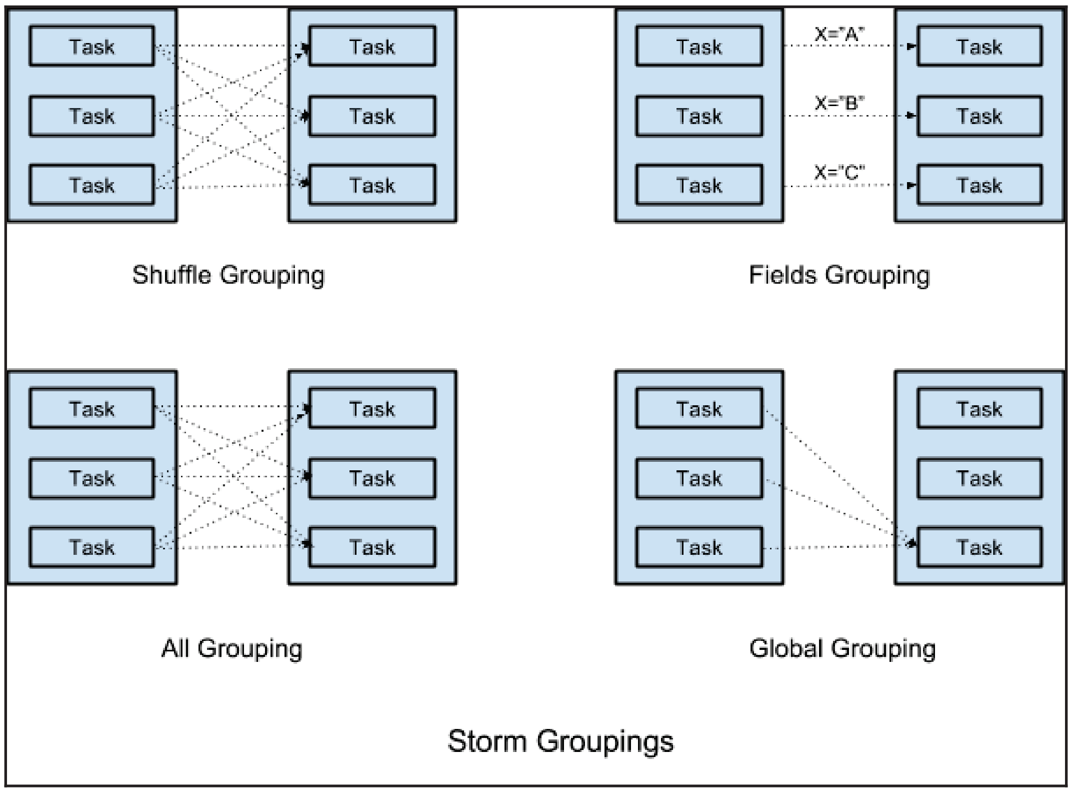
Custom grouping

If none of the preceding groupings fit your use case, you can define your own custom grouping by implementing the `backtype.storm.grouping.CustomStreamGrouping` interface.

Here is a sample custom grouping that partitions the stream on the basis of the category in the tuples:

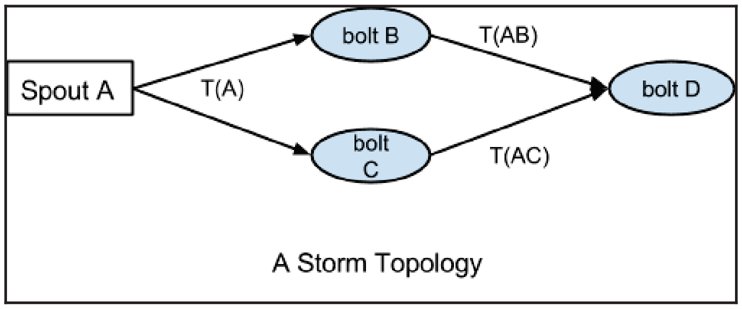
```
public class CategoryGrouping implements CustomStreamGrouping, Serializable
{
  private static final Map<String, Integer> categories = ImmutableMap.of
  (
    "Financial", 0,
    "Medical", 1,
    "FMCG", 2,
    "Electronics", 3
  );

  private int tasks = 0;

  public void prepare(WorkerTopologyContext context, GlobalStreamId stream,
List<Integer> targetTasks)
  {
    tasks = targetTasks.size();
  }

  public List<Integer> chooseTasks(int taskId, List<Object> values) {
    String category = (String) values.get(0);
    return ImmutableList.of(categories.get(category) % tasks);
  }
}
```

The following diagram represents the Storm groupings graphically:



Guaranteed message processing

In a Storm topology, a single tuple being emitted by a spout can result in a number of tuples being generated in the later stages of the topology. For example, consider the following topology:



Here, **Spout A** emits a tuple **T(A)**, which is processed by **bolt B** and **bolt C**, which emit tuple **T(AB)** and **T(AC)** respectively. So, when all the tuples produced as a result of tuple **T(A)**--namely, the tuple tree **T(A)**, **T(AB)**, and **T(AC)**--are processed, we say that the tuple has been processed completely.

When some of the tuples in a tuple tree fail to process either due to some runtime error or a timeout that is configurable for each topology, then Storm considers that to be a failed tuple.

Here are the six steps that are required by Storm to guarantee message processing:

1. Tag each tuple emitted by a spout with a unique message ID. This can be done by using the `org.apache.storm.spout.SpoutOutputColletor.emit` method, which takes a `messageId` argument. Storm uses this message ID to track the state of the tuple tree generated by this tuple. If you use one of the emit methods that doesn't take a `messageId` argument, Storm will not track it for complete processing. When the message is processed completely, Storm will send an acknowledgement with the same `messageId` that was used while emitting the tuple.
2. A generic pattern implemented by spouts is that they read a message from a messaging queue, say RabbitMQ, produce the tuple into the topology for further processing, and then dequeue the message once it receives the acknowledgement that the tuple has been processed completely.
3. When one of the bolts in the topology needs to produce a new tuple in the course of processing a message, for example, **bolt B** in the preceding topology, then it should emit the new tuple anchored with the original tuple that it got from the spout. This can be done by using the overloaded emit methods in the `org.apache.storm.task.OutputCollector` class that takes an anchor tuple as an argument. If you are emitting multiple tuples from the same input tuple, then anchor each outgoing tuple.

4. Whenever you are done with processing a tuple in the execute method of your bolt, send an acknowledgment using the `org.apache.storm.task.OutputCollector.ack` method. When the acknowledgement reaches the emitting spout, you can safely mark the message as being processed and dequeue it from the message queue, if any.
5. Similarly, if there is some problem in processing a tuple, a failure signal should be sent back using the `org.apache.storm.task.OutputCollector.fail` method so that Storm can replay the failed message.
6. One of the general patterns of processing in Storm bolts is to process a tuple in, emit new tuples, and send an acknowledgement at the end of the execute method. Storm provides the `org.apache.storm.topology.base.BasicBasicBolt` class that automatically sends the acknowledgement at the end of the execute method. If you want to signal a failure, throw `org.apache.storm.topology.FailedException` from the execute method.

This model results in at-least-once message processing semantics, and your application should be ready to handle a scenario when some of the messages will be processed multiple times. Storm also provides exactly-once message processing semantics, which we will discuss in `Chapter 5`, *Trident Topology and Uses*.

Even though you can achieve some guaranteed message processing in Storm using the methods mentioned here, it is always a point to ponder whether you actually require it or not, as you can gain a large performance boost by risking some of the messages not being completely processed by Storm. This is a trade-off that you can think of when designing your application.

Tick tuple

In some use cases, a bolt needs to cache the data for a few seconds before performing some operation, such as cleaning the cache after every 5 seconds or inserting a batch of records into a database in a single request.

The tick tuple is the system-generated (Storm-generated) tuple that we can configure at each bolt level. The developer can configure the tick tuple at the code level while writing a bolt.

We need to overwrite the following method in the bolt to enable the tick tuple:

```
@Override
public Map<String, Object> getComponentConfiguration() {
  Config conf = new Config();
  int tickFrequencyInSeconds = 10;
  conf.put(Config.TOPOLOGY_TICK_TUPLE_FREQ_SECS,
  tickFrequencyInSeconds);
  return conf;
}
```

In the preceding code, we have configured the tick tuple time to 10 seconds. Now, Storm will start generating a tick tuple after every 10 seconds.

Also, we need to add the following code in the execute method of the bolt to identify the type of tuple:

```
@Override
public void execute(Tuple tuple) {
  if (isTickTuple(tuple)) {
    // now you can trigger e.g. a periodic activity
  }
  else {
    // do something with the normal tuple
  }
}

private static boolean isTickTuple(Tuple tuple) {
  return
  tuple.getSourceComponent().equals(Constants.SYSTEM_COMPONENT_ID) &&
tuple.getSourceStreamId().equals(Constants.SYSTEM_TICK_STREAM_ID);
}
```

If the output of the `isTickTuple()` method is true, then the input tuple is a tick tuple. Otherwise, it is a normal tuple emitted by the previous bolt.

Be aware that tick tuples are sent to bolts/spouts just like regular tuples, which means they will be queued behind other tuples that a bolt/spout is about to process via its `execute()` or `nextTuple()` method, respectively. As such, the time interval you configure for tick tuples is, in practice, served on a best-effort basis. For instance, if a bolt is suffering from high execution latency--for example, due to being overwhelmed by the incoming rate of regular, non-tick tuples--then you will observe that the periodic activities implemented in the bolt will get triggered later than expected.

Summary

In this chapter, we have shed some light on how we can define the parallelism of Storm, how we can distribute jobs between multiple nodes, and how we can distribute data between multiple instances of a bolt. The chapter also covered two important features: guaranteed message processing and the tick tuple.

In the next chapter, we are covering the Trident high-level abstraction over Storm. Trident is mostly used to solve the real-time transaction problem, which can't be solved through plain Storm.

8
Integration of Storm and Kafka

Apache Kafka is a high-throughput, distributed, fault-tolerant, and replicated messaging system that was first developed at LinkedIn. The use cases of Kafka vary from log aggregation, to stream processing, to replacing other messaging systems.

Kafka has emerged as one of the important components of real-time processing pipelines in combination with Storm. Kafka can act as a buffer or feeder for messages that need to be processed by Storm. Kafka can also be used as the output sink for results emitted from Storm topologies.

In this chapter, we will be covering the following topics:

- Kafka architecture--broker, producer, and consumer
- Installation of the Kafka cluster
- Sharing the producer and consumer between Kafka
- Development of Storm topology using Kafka consumer as Storm spout
- Deployment of a Kafka and Storm integration topology

Introduction to Kafka

In this section we are going to cover the architecture of Kafka--broker, consumer, and producer.

Kafka architecture

Kafka has an architecture that differs significantly from other messaging systems. Kafka is a peer to peer system (each node in a cluster has the same role) in which each node is called a **broker**. The brokers coordinate their actions with the help of a ZooKeeper ensemble. The Kafka metadata managed by the ZooKeeper ensemble is mentioned in the section *Sharing ZooKeeper between Storm and Kafka*:

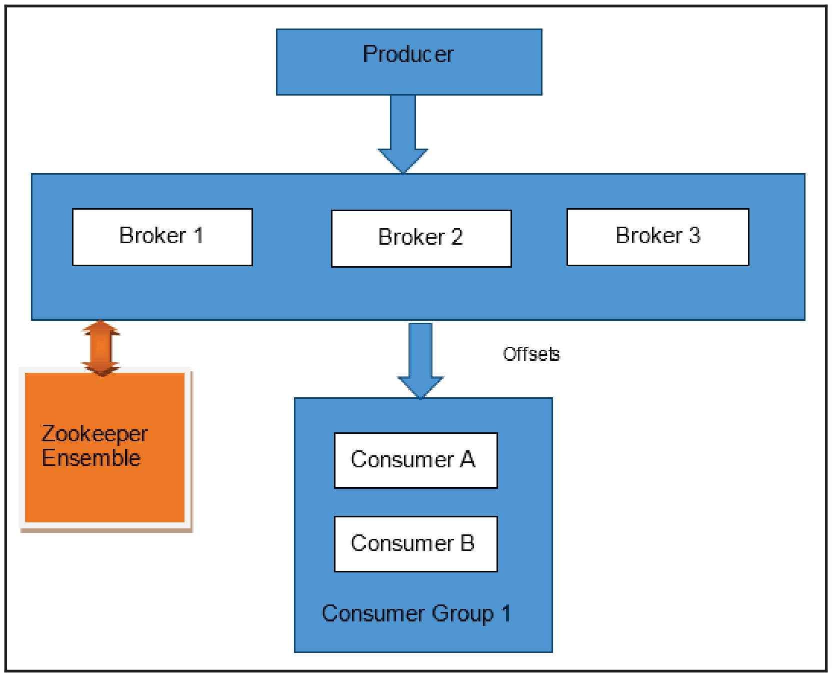


Figure 8.1: A Kafka cluster

The following are the important components of Kafka:

Producer

A producer is an entity that uses the Kafka client API to publish messages into the Kafka cluster. In a Kafka broker, messages are published by the producer entity to named entities called **topics**. A topic is a persistent queue (data stored into topics is persisted to disk).

For parallelism, a Kafka topic can have multiple partitions. Each partition data is represented in a different file. Also, two partitions of a single topic can be allocated on a different broker, thus increasing throughput as all partitions are independent of each other. Each message in a partition has a unique sequence number associated with it called an **offset**:

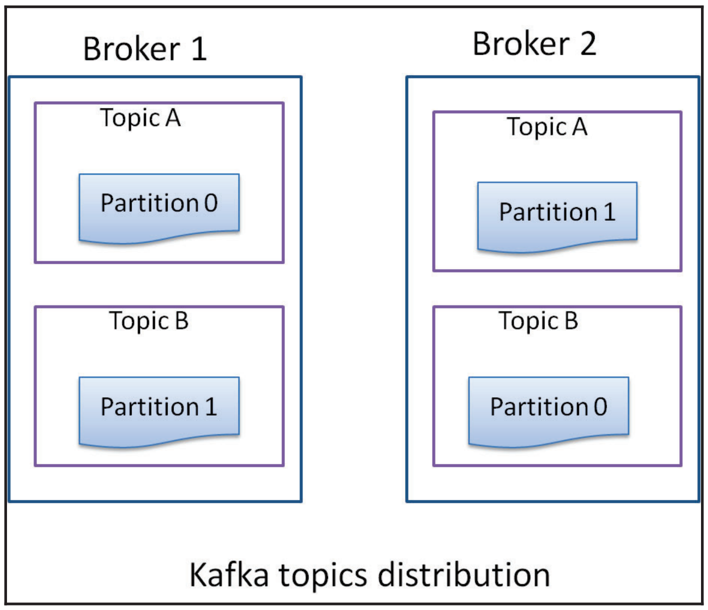


Figure 8.2: Kafka topic distribution

Replication

Kafka supports the replication of partitions of topics to support fault tolerance. Kafka automatically handles the replication of partitions and makes sure that the replica of a partition will be assigned to different brokers. Kafka elects one broker as the leader of a partition and all writes and reads must go to the partition leader. Replication features are introduced in Kafka 8.0.0 version.

The Kafka cluster manages the list of **in sync replica** (**ISR**)--the replicate which are in sync with the partition leader into ZooKeeper. If the partition leader goes down, then the followers/replicas that are present in the ISR list are only eligible for the next leader of the failed partition.

Consumer

Consumers read a range of messages from a broker. Each consumer has an assigned group ID. All the consumers with the same group ID act as a single logical consumer. Each message of a topic is delivered to one consumer from a consumer group (with the same group ID). Different consumer groups for a particular topic can process messages at their own pace as messages are not removed from the topics as soon as they are consumed. In fact, it is the responsibility of the consumers to keep track of how many messages they have consumed.

As mentioned earlier, each message in a partition has a unique sequence number associated with it called an offset. It is through this offset that consumers know how much of the stream they have already processed. If a consumer decides to replay already processed messages, all it needs to do is just set the value of an offset to an earlier value before consuming messages from Kafka.

Broker

The broker receives the messages from the producer (push mechanism) and delivers the messages to the consumer (pull mechanism). Brokers also manage the persistence of messages in a file. Kafka brokers are very lightweight: they only open file pointers on a queue (topic partitions) and manage TCP connections.

Data retention

Each topic in Kafka has an associated retention time. When this time expires, Kafka deletes the expired data file for that particular topic. This is a very efficient operation as it's a file delete operation.

Installation of Kafka brokers

At the time of writing, the stable version of Kafka is 0.9.x.

The prerequisites for running Kafka are a ZooKeeper ensemble and Java Version 1.7 or above. Kafka comes with a convenience script that can start a single node ZooKeeper but it is not recommended to use it in a production environment. We will be using the ZooKeeper cluster we deployed in `Chapter 2`, *Storm Deployment, Topology Development, and Topology Options*.

We will see how to set up a single node Kafka cluster first and then how to add two more nodes to it to run a full-fledged, three node Kafka cluster with replication enabled.

Setting up a single node Kafka cluster

Following are the steps to set up a single node Kafka cluster:

1. Download the Kafka 0.9.x binary distribution named `kafka_2.10-0.9.0.1.tar.gz` from `http://apache.claz.org/kafka/0.9.0.1/kafka_2.10-0.9.0.1.tgz`.

2. Extract the archive to wherever you want to install Kafka with the following command:

   ```
   tar -xvzf kafka_2.10-0.9.0.1.tgz
   cd kafka_2.10-0.9.0.1
   ```

   We will refer to the Kafka installation directory as `$KAFKA_HOME` from now on.

3. Change the following properties in the `$KAFKA_HOME/config/server.properties` file:

   ```
   log.dirs=/var/kafka-
   logszookeeper.connect=zoo1:2181,zoo2:2181,zoo3:2181
   ```

   Here, `zoo1`, `zoo2`, and `zoo3` represent the hostnames of the ZooKeeper nodes.

   The following are the definitions of the important properties in the `server.properties` file:

   - `broker.id`: This is a unique integer ID for each of the brokers in a Kafka cluster.
   - `port`: This is the port number for a Kafka broker. Its default value is `9092`. If you want to run multiple brokers on a single machine, give a unique port to each broker.
   - `host.name`: The hostname to which the broker should bind and advertise itself.

- `log.dirs`: The name of this property is a bit unfortunate as it represents not the log directory for Kafka, but the directory where Kafka stores the actual data sent to it. This can take a single directory or a comma-separated list of directories to store data. Kafka throughput can be increased by attaching multiple physical disks to the broker node and specifying multiple data directories, each lying on a different disk. It is not much use specifying multiple directories on the same physical disk, as all the I/O will still be happening on the same disk.
- `num.partitions`: This represents the default number of partitions for newly created topics. This property can be overridden when creating new topics. A greater number of partitions results in greater parallelism at the cost of a larger number of files.
- `log.retention.hours`: Kafka does not delete messages immediately after consumers consume them. It retains them for the number of hours defined by this property so that in the event of any issues the consumers can replay the messages from Kafka. The default value is `168` hours, which is 1 week.
- `zookeeper.connect`: This is the comma-separated list of ZooKeeper nodes in `hostname:port` form.

4. Start the Kafka server by running the following command:

```
> ./bin/kafka-server-start.sh config/server.properties
[2017-04-23 17:44:36,667] INFO New leader is 0
(kafka.server.ZookeeperLeaderElector$LeaderChangeListener)
[2017-04-23 17:44:36,668] INFO Kafka version : 0.9.0.1
(org.apache.kafka.common.utils.AppInfoParser)
[2017-04-23 17:44:36,668] INFO Kafka commitId : a7a17cdec9eaa6c5
(org.apache.kafka.common.utils.AppInfoParser)
[2017-04-23 17:44:36,670] INFO [Kafka Server 0], started
(kafka.server.KafkaServer)
```

If you get something similar to the preceding three lines on your console, then your Kafka broker is up-and-running and we can proceed to test it.

5. Now we will verify that the Kafka broker is set up correctly by sending and receiving some test messages. First, let's create a verification topic for testing by executing the following command:

   ```
   > bin/kafka-topics.sh --zookeeper zoo1:2181 --replication-factor 1
   --partition 1 --topic verification-topic --create
   Created topic "verification-topic".
   ```

6. Now let's verify if the topic creation was successful by listing all the topics:

   ```
   > bin/kafka-topics.sh --zookeeper zoo1:2181 --list
   verification-topic
   ```

7. The topic is created; let's produce some sample messages for the Kafka cluster. Kafka comes with a command-line producer that we can use to produce messages:

   ```
   > bin/kafka-console-producer.sh --broker-list localhost:9092 --
   topic verification-topic
   ```

8. Write the following messages on your console:

   ```
   Message 1
   Test Message 2
   Message 3
   ```

9. Let's consume these messages by starting a new console consumer on a new console window:

   ```
   > bin/kafka-console-consumer.sh --zookeeper localhost:2181 --topic
   verification-topic --from-beginning
   Message 1
   Test Message 2
   Message 3
   ```

Now, if we enter any message on the producer console, it will automatically be consumed by this consumer and displayed on the command line.

Using Kafka's single node ZooKeeper
If you don't want to use an external ZooKeeper ensemble, you can use the single node ZooKeeper instance that comes with Kafka for quick and dirty development. To start using it, first modify the `$KAFKA_HOME/config/zookeeper.properties` file to specify the data directory by supplying following property:
`dataDir=/var/zookeeper`
Now, you can start the Zookeeper instance with the following command:
```
> ./bin/zookeeper-server-start.sh
config/zookeeper.properties
```

Setting up a three node Kafka cluster

So far we have a single node Kafka cluster. Follow the steps to deploy the Kafka cluster:

1. Create a three node VM or three physical machines.
2. Perform steps 1 and 2 mentioned in the section *Setting up a single node Kafka cluster*.
3. Change the following properties in the file `$KAFKA_HOME/config/server.properties`:

   ```
   broker.id=0
   port=9092
   host.name=kafka1
   log.dirs=/var/kafka-logs
   zookeeper.connect=zoo1:2181,zoo2:2181,zoo3:2181
   ```

   Make sure that the value of the `broker.id` property is unique for each Kafka broker and the value of `zookeeper.connect` must be the same on all nodes.

4. Start the Kafka brokers by executing the following command on all three boxes:

   ```
   > ./bin/kafka-server-start.sh config/server.properties
   ```

5. Now let's verify the setup. First we create a topic with the following command:

   ```
   > bin/kafka-topics.sh --zookeeper zoo1:2181 --replication-factor 4
   --partition 1 --topic verification --create
       Created topic "verification-topic".
   ```

6. Now, we will list the topics to see if the topic was created successfully:

   ```
   > bin/kafka-topics.sh --zookeeper zoo1:2181 --list
                   topic: verification     partition: 0      leader: 0
   replicas: 0             isr: 0
                   topic: verification     partition: 1      leader: 1
   replicas: 1             isr: 1
                   topic: verification     partition: 2      leader: 2
   replicas: 2             isr: 2
   ```

7. Now, we will verify the setup by using the Kafka console producer and consumer as done in the *Setting up a single node Kafka cluster* section:

   ```
   > bin/kafka-console-producer.sh --broker-list
   kafka1:9092,kafka2:9092,kafka3:9092 --topic verification
   ```

8. Write the following messages on your console:

   ```
   First
   Second
   Third
   ```

9. Let's consume these messages by starting a new console consumer on a new console window:

   ```
   > bin/kafka-console-consumer.sh --zookeeper localhost:2181 --topic
   verification --from-beginning
   First
   Second
   Third
   ```

So far, we have three brokers on the Kafka cluster working. In the next section, we will see how to write a producer that can produce messages to Kafka:

Multiple Kafka brokers on a single node

If you want to run multiple Kafka brokers on a single node, then follow the following steps:

1. Copy `config/server.properties` to create `config/server1.properties` and `config/server2.properties`.

2. Populate the following properties in `config/server.properties`:

   ```
   broker.id=0
   port=9092
   log.dirs=/var/kafka-logs
   zookeeper.connect=zoo1:2181,zoo2:2181,zoo3:2181
   ```

3. Populate the following properties in `config/server1.properties`:

   ```
   broker.id=1
   port=9093
   log.dirs=/var/kafka-1-logs
   zookeeper.connect=zoo1:2181,zoo2:2181,zoo3:2181
   ```

4. Populate the following properties in `config/server2.properties`:

   ```
   broker.id=2
   port=9094
   log.dirs=/var/kafka-2-logs
   zookeeper.connect=zoo1:2181,zoo2:2181,zoo3:2181
   ```

5. Run the following commands on three different terminals to start Kafka brokers:

   ```
   > ./bin/kafka-server-start.sh config/server.properties
   > ./bin/kafka-server-start.sh config/server1.properties
   > ./bin/kafka-server-start.sh config/server2.properties
   ```

Share ZooKeeper between Storm and Kafka

We can share the same ZooKeeper ensemble between Kafka and Storm as both store the metadata inside the different znodes (ZooKeeper coordinates between the distributed processes using the shared hierarchical namespace, which is organized similarly to a standard file system. In ZooKeeper, the namespace consisting of data registers is called znodes).

We need to open the ZooKeeper client console to view the znodes (shared namespace) created for Kafka and Storm.

Go to `ZK_HOME` and execute the following command to open the ZooKeeper console:

```
> bin/zkCli.sh
```

Execute the following command to view the list of znodes:

```
> [zk: localhost:2181(CONNECTED) 0] ls /
[storm, consumers, isr_change_notification, zookeeper, admin, brokers]
```

Here, consumers, `isr_change_notification`, and brokers are the znodes and the Kafka is managing its metadata information into ZooKeeper at this location.

Storm manages its metadata inside the Storm znodes in ZooKeeper.

Kafka producers and publishing data into Kafka

In this section we are writing a Kafka producer that will publish events into the Kafka topic.

Perform the following step to create the producer:

1. Create a Maven project by using `com.stormadvance` as `groupId` and `kafka-producer` as `artifactId`.
2. Add the following dependencies for Kafka in the `pom.xml` file:

```
<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka_2.10</artifactId>
  <version>0.9.0.1</version>
  <exclusions>
    <exclusion>
      <groupId>com.sun.jdmk</groupId>
      <artifactId>jmxtools</artifactId>
    </exclusion>
    <exclusion>
      <groupId>com.sun.jmx</groupId>
      <artifactId>jmxri</artifactId>
    </exclusion>
  </exclusions>
</dependency>
<dependency>
  <groupId>org.apache.logging.log4j</groupId>
  <artifactId>log4j-slf4j-impl</artifactId>
  <version>2.0-beta9</version>
</dependency>
<dependency>
  <groupId>org.apache.logging.log4j</groupId>
  <artifactId>log4j-1.2-api</artifactId>
```

```
    <version>2.0-beta9</version>
  </dependency>
```

3. Add the following `build` plugins to the `pom.xml` file. It will let us execute the producer using Maven:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>exec-maven-plugin</artifactId>
      <version>1.2.1</version>
      <executions>
        <execution>
          <goals>
            <goal>exec</goal>
          </goals>
        </execution>
      </executions>
      <configuration>
        <executable>java</executable
<includeProjectDependencies>true</includeProjectDependencies
<includePluginDependencies>false</includePluginDependencies>
        <classpathScope>compile</classpathScope>
        <mainClass>com.stormadvance.kafka_producer.
KafkaSampleProducer
        </mainClass>
      </configuration>
    </plugin>
  </plugins>
</build>
```

4. Now we will create the `KafkaSampleProducer` class in the `com.stormadvance.kafka_producer` package. This class will produce each word from the first paragraph of Franz Kafka's Metamorphosis into the `new_topic` topic in Kafka as single message. The following is the code for the `KafkaSampleProducer` class with explanations:

```
public class KafkaSampleProducer {
  public static void main(String[] args) {
    // Build the configuration required for connecting to Kafka
    Properties props = new Properties();

    // List of kafka borkers. Complete list of brokers is not
required as
    // the producer will auto discover the rest of the brokers.
    props.put("bootstrap.servers", "Broker1-IP:9092");
    props.put("batch.size", 1);
    // Serializer used for sending data to kafka. Since we are
sending string,
    // we are using StringSerializer.
    props.put("key.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
    props.put("value.serializer",
"org.apache.kafka.common.serialization.StringSerializer");

    props.put("producer.type", "sync");
    // Create the producer instance
    Producer<String, String> producer = new KafkaProducer<String,
String>(props);

    // Now we break each word from the paragraph
    for (String word : METAMORPHOSIS_OPENING_PARA.split("\\s")) {
      System.out.println("word : " + word);
      // Create message to be sent to "new_topic" topic with the
word
      ProducerRecord<String, String> data = new
ProducerRecord<String, String>("new_topic",word, word);
      // Send the message
      producer.send(data);
    }

    // close the producer
    producer.close();
    System.out.println("end : ");
  }

  // First paragraph from Franz Kafka's Metamorphosis
  private static String METAMORPHOSIS_OPENING_PARA = "One morning,
when Gregor Samsa woke from troubled dreams, he found "
               + "himself transformed in his bed into a horrible
vermin.  He lay on "
               + "his armour-like back, and if he lifted his head a
little he could "
               + "see his brown belly, slightly domed and divided
by arches into stiff "
               + "sections.  The bedding was hardly able to cover
it and seemed ready "
               + "to slide off any moment.  His many legs,
pitifully thin compared "
               + "with the size of the rest of him, waved about
helplessly as he "
               + "looked.";

}
```

5. Now, before running the producer, we need to create `new_topic` in Kafka. To do so, execute the following command:

```
> bin/kafka-topics.sh --zookeeper ZK1:2181 --replication-factor 1 -
-partition 1 --topic new_topic --create
Created topic "new_topic1".
```

6. Now we can run the producer by executing the following command:

```
> mvn compile exec:java
......
103  [com.learningstorm.kafka.WordsProducer.main()] INFO
kafka.client.ClientUti
ls$  - Fetching metadata from broker
id:0,host:kafka1,port:9092 with correlation id 0 for 1
topic(s) Set(words_topic)
110  [com.learningstorm.kafka.WordsProducer.main()] INFO
kafka.producer.SyncProducer  - Connected to kafka1:9092 for
producing
140  [com.learningstorm.kafka.WordsProducer.main()] INFO
kafka.producer.SyncProducer  - Disconnecting from
kafka1:9092
177  [com.learningstorm.kafka.WordsProducer.main()] INFO
kafka.producer.SyncProducer  - Connected to kafka1:9092 for
producing
378  [com.learningstorm.kafka.WordsProducer.main()] INFO
kafka.producer.Producer  - Shutting down producer
378  [com.learningstorm.kafka.WordsProducer.main()] INFO
kafka.producer.ProducerPool  - Closing all sync producers
381  [com.learningstorm.kafka.WordsProducer.main()] INFO
kafka.producer.SyncProducer  - Disconnecting from
kafka1:9092
```

7. Now let us verify that the message has been produced by using Kafka's console consumer and executing the following command:

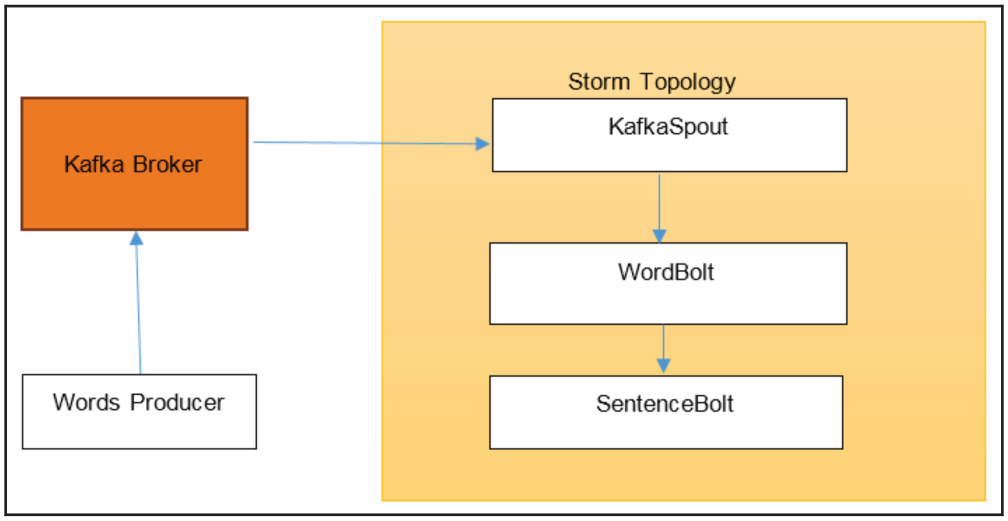
```
> bin/kafka-console-consumer.sh --zookeeper ZK:2181 --topic
verification --from-beginning
                One
                morning,
                when
                Gregor
                Samsa
                woke
                from
                troubled
                dreams,
                he
                found
                himself
                transformed
                in
                his
                bed
                into
                a
                horrible
                vermin.
                ......
```

So, we are able to produce messages into Kafka. In the next section, we will see how we can use `KafkaSpout` to read messages from Kafka and process them inside a Storm topology.

Kafka Storm integration

Now we will create a Storm topology that will consume messages from the Kafka topic `new_topic` and aggregate words into sentences.

The complete message flow is shown as follows:



We have already seen `KafkaSampleProducer`, which produces words into the Kafka broker. Now we will create a Storm topology that will read those words from Kafka to aggregate them into sentences. For this, we will have one `KafkaSpout` in the application that will read the messages from Kafka and two bolts, `WordBolt` that receive words from `KafkaSpout` and then aggregate them into sentences, which are then passed onto the `SentenceBolt`, which simply prints them on the output stream. We will be running this topology in a local mode.

Follow the steps to create the Storm topology:

1. Create a new Maven project with `groupId` as `com.stormadvance` and `artifactId` as `kafka-storm-topology`.
2. Add the following dependencies for Kafka-Storm and Storm in the `pom.xml` file:

```
<dependency>
  <groupId>org.apache.storm</groupId>
  <artifactId>storm-kafka</artifactId>
  <version>1.0.2</version>
  <exclusions>
    <exclusion>
      <groupId>org.apache.kafka</groupId>
      <artifactId>kafka-clients</artifactId>
    </exclusion>
  </exclusions>
</dependency>

<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka_2.10</artifactId>
  <version>0.9.0.1</version>
  <exclusions>
    <exclusion>
      <groupId>com.sun.jdmk</groupId>
      <artifactId>jmxtools</artifactId>
    </exclusion>
    <exclusion>
      <groupId>com.sun.jmx</groupId>
      <artifactId>jmxri</artifactId>
    </exclusion>
  </exclusions>
</dependency>

<dependency>
  <groupId>org.apache.storm</groupId>
  <artifactId>storm-core</artifactId>
  <version>1.0.2</version>
```

```
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>commons-collections</groupId>
    <artifactId>commons-collections</artifactId>
    <version>3.2.1</version>
  </dependency>

  <dependency>
    <groupId>com.google.guava</groupId>
    <artifactId>guava</artifactId>
    <version>15.0</version>
  </dependency>
```

3. Add the following Maven plugins to the `pom.xml` file so that we are able to run it from the command-line and also to package the topology to be executed in Storm:

```
<build>
  <plugins>
    <plugin>
      <artifactId>maven-assembly-plugin</artifactId>
      <configuration>
        <descriptorRefs>
          descriptorRef>jar-with-dependencies</descriptorRef>
        </descriptorRefs>
        <archive>
          <manifest>
            <mainClass></mainClass>
          </manifest>
        </archive>
      </configuration>
      <executions>
        <execution>
          <id>make-assembly</id>
          <phase>package</phase>
          <goals>
            <goal>single</goal>
          </goals>
        </execution>
      </executions>
    </plugin>

    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>exec-maven-plugin</artifactId>
      <version>1.2.1</version>
      <executions>
        <execution>
```

```
              <goals>
                <goal>exec</goal>
              </goals>
            </execution>
          </executions>
          <configuration>
            <executable>java</executable
<includeProjectDependencies>true</includeProjectDependencies
<includePluginDependencies>false</includePluginDependencies>
            <classpathScope>compile</classpathScope>
            <mainClass>${main.class}</mainClass>
          </configuration>
        </plugin>

        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-compiler-plugin</artifactId>
        </plugin>

      </plugins>
    </build>
```

4. Now we will first create the `WordBolt` that will aggregate the words into sentences. For this, create a class called `WordBolt` in the `com.stormadvance.kafka` package. The code for `WordBolt` is as follows, complete with explanation:

```
public class WordBolt extends BaseBasicBolt {

  private static final long serialVersionUID =
-5353547217135922477L;

  // list used for aggregating the words
  private List<String> words = new ArrayList<String>();

  public void execute(Tuple input, BasicOutputCollector collector)
{
    System.out.println("called");
    // Get the word from the tuple
    String word = input.getString(0);

    if (StringUtils.isBlank(word)) {
      // ignore blank lines
      return;
    }

    System.out.println("Received Word:" + word);
```

```
        // add word to current list of words
        words.add(word);

        if (word.endsWith(".")) {
          // word ends with '.' which means this is // the end of the
    sentence
          // publish a sentence tuple
          collector.emit(ImmutableList.of((Object)
    StringUtils.join(words, ' ')));

          // reset the words list.
          words.clear();
        }
      }

      public void declareOutputFields(OutputFieldsDeclarer declarer) {
        // here we declare we will be emitting tuples with
        // a single field called "sentence"
        declarer.declare(new Fields("sentence"));
      }
    }
```

5. Next is `SentenceBolt`, which just prints the sentences that it receives. Create `SentenceBolt` in the `com.stormadvance.kafka` package. The code is as follows, with explanations:

```
    public class SentenceBolt extends BaseBasicBolt {

      private static final long serialVersionUID =
    7104400131657100876L;

      public void execute(Tuple input, BasicOutputCollector collector)
    {
        // get the sentence from the tuple and print it
        System.out.println("Recieved Sentence:");
        String sentence = input.getString(0);
        System.out.println("Recieved Sentence:" + sentence);
      }

      public void declareOutputFields(OutputFieldsDeclarer declarer) {
             // we don't emit anything
      }
    }
```

6. Now we will create the `KafkaTopology` that will define the `KafkaSpout` and wire it with `WordBolt` and `SentenceBolt`. Create a new class called `KafkaTopology` in the `com.stormadvance.kafka` package. The code is as follows, with explanations:

```
public class KafkaTopology {
  public static void main(String[] args) {
    try {
      // ZooKeeper hosts for the Kafka cluster
      BrokerHosts zkHosts = new ZkHosts("ZKIP:PORT");

      // Create the KafkaSpout configiuartion
      // Second argument is the topic name
      // Third argument is the zookeepr root for Kafka
      // Fourth argument is consumer group id
      SpoutConfig kafkaConfig = new SpoutConfig(zkHosts,
"new_topic", "", "id1");

      // Specify that the kafka messages are String
      // We want to consume all the first messages in the topic
everytime
      // we run the topology to help in debugging. In production,
this
      // property should be false
      kafkaConfig.scheme = new SchemeAsMultiScheme(new
StringScheme());
      kafkaConfig.startOffsetTime =
kafka.api.OffsetRequest.EarliestTime();

      // Now we create the topology
      TopologyBuilder builder = new TopologyBuilder();

      // set the kafka spout class
      builder.setSpout("KafkaSpout", new KafkaSpout(kafkaConfig),
2);

      // set the word and sentence bolt class
      builder.setBolt("WordBolt", new WordBolt(),
1).globalGrouping("KafkaSpout");
      builder.setBolt("SentenceBolt", new SentenceBolt(),
1).globalGrouping("WordBolt");

      // create an instance of LocalCluster class for executing
topology
      // in local mode.
      LocalCluster cluster = new LocalCluster();
      Config conf = new Config();
      conf.setDebug(true);
      if (args.length > 0) {
        conf.setNumWorkers(2);
        conf.setMaxSpoutPending(5000);
        StormSubmitter.submitTopology("KafkaToplogy1", conf,
builder.createTopology());

      } else {
        // Submit topology for execution
        cluster.submitTopology("KafkaToplogy1", conf,
builder.createTopology());
        System.out.println("called1");
        Thread.sleep(1000000);
        // Wait for sometime before exiting
        System.out.println("Waiting to consume from kafka");

        System.out.println("called2");
        // kill the KafkaTopology
        cluster.killTopology("KafkaToplogy1");
        System.out.println("called3");
        // shutdown the storm test cluster
        cluster.shutdown();
      }

    } catch (Exception exception) {
      System.out.println("Thread interrupted exception : " +
exception);
    }
  }
}
```

7. Now we will the run the topology. Make sure the Kafka cluster is running and you have executed the producer in the last section so that there are messages in Kafka for consumption.
8. Run the topology by executing the following command:

```
> mvn clean compile exec:java -
Dmain.class=com.stormadvance.kafka.KafkaTopology
```

This will execute the topology. You should see messages similar to the following in your output:

```
Recieved Word:One
Recieved Word:morning,
Recieved Word:when
Recieved Word:Gregor
Recieved Word:Samsa
Recieved Word:woke
Recieved Word:from
Recieved Word:troubled
Recieved Word:dreams,
Recieved Word:he
Recieved Word:found
Recieved Word:himself
Recieved Word:transformed
Recieved Word:in
Recieved Word:his
Recieved Word:bed
Recieved Word:into
Recieved Word:a
Recieved Word:horrible
Recieved Word:vermin.
Recieved Sentence:One morning, when Gregor Samsa woke from
troubled dreams, he found himself transformed in his bed
into a horrible vermin.
```

So we are able to consume messages from Kafka and process them in a Storm topology.

Deploy the Kafka topology on Storm cluster

The deployment of Kafka and Storm integration topology on the Storm cluster is similar to the deployment of other topologies. We need to set the number of workers and the maximum spout pending Storm config and we need to use the `submitTopology` method of `StormSubmitter` to submit the topology on the Storm cluster.

Now, we need to build the topology code as mentioned in the following steps to create a JAR of the Kafka Storm integration topology:

1. Go to project home.
2. Execute the command:

   ```
   mvn clean install
   ```

   The output of the preceding command is as follows:

   ```
   ------------------------------------------------------------------
   -----
   [INFO] -----------------------------------------------------------
   -----
   [INFO] BUILD SUCCESS
   [INFO] -----------------------------------------------------------
   -----
   ```

```
[INFO] Total time: 58.326s
[INFO] Finished at:
[INFO] Final Memory: 14M/116M
[INFO] ------------------------------------------------------------
-----
```

3. Now, copy the Kafka Storm topology on the Nimbus machine and execute the following command to submit the topology on the Storm cluster:

   ```
   bin/storm jar jarName.jar [TopologyMainClass] [Args]
   ```

   The preceding command runs `TopologyMainClass` with the argument. The main function of `TopologyMainClass` is to define the topology and submit it to Nimbus. The Storm JAR part takes care of connecting to Nimbus and uploading the JAR part.

4. Log in on the Storm Nimbus machine and execute the following commands:

   ```
   $> cd $STORM_HOME
   $> bin/storm jar ~/storm-kafka-topology-0.0.1-SNAPSHOT-jar-with-
   dependencies.jar com.stormadvance.kafka.KafkaTopology
   KafkaTopology1
   ```

   Here, `~/ storm-kafka-topology-0.0.1-SNAPSHOT-jar-with-dependencies.jar` is the path of the `KafkaTopology` JAR we are deploying on the Storm cluster.

Summary

In this chapter, we learned about the basics of Apache Kafka and how to use it as part of a real-time stream processing pipeline build with Storm. We learned about the architecture of Apache Kafka and how it can be integrated into Storm processing by using `KafkaSpout`.

In the next chapter, we are going to cover the integration of Storm with Hadoop and YARN. We are also going to cover sample examples for this operation.