Reference Architectures 2017
Red Hat JBoss Data Grid 7 and Apache Spark

Calvin Zhu

Reference Architectures 2017 Red Hat JBoss Data Grid 7 and Apache Spark

Calvin Zhu
refarch-feedback@redhat.com

Legal Notice

Copyright © 2017 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at
http://creativecommons.org/licenses/by-sa/3.0/
. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat Software Collections is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Abstract

This reference architecture demonstrate an example usage of Red Hat JBoss Data Grid 7.0 with an Apache Spark 1.6 cluster. In addressing a hypothetical use case for the internet of things (IoT), a JDG 7 cluster ingests and serves data from IoT sensors while also acting as a highly scalable, high-performance data source for the Apache Spark cluster. The paper aims to provide a thorough description of the steps required for using Apache Spark with JDG 7 through the new Spark connector, and how to set up a cluster environment for JDG 7 and Spark.

Table of Contents

COMMENTS AND FEEDBACK 3

CHAPTER 1. EXECUTIVE SUMMARY 4

CHAPTER 2. REFERENCE ARCHITECTURE ENVIRONMENT 5
- 2.1. APACHE SPARK 5
- 2.2. RED HAT JBOSS DATA GRID 7 5

CHAPTER 3. CREATING THE ENVIRONMENT 7
- 3.1. PREREQUISITES 7
- 3.2. DOWNLOADS 7
- 3.3. INSTALLATION 7
- 3.4. CONFIGURATION 7
- 3.5. STARTUP 11

CHAPTER 4. SENSOR APPLICATION 15
- 4.1. OVERVIEW 15
- 4.2. PACKAGES OVERVIEW 15
- 4.3. SENSOR PACKAGE 17
- 4.4. ANALYZER PACKAGE 20
- 4.5. CLIENT PACKAGE 25

APPENDIX A. REVISION HISTORY 30

APPENDIX B. CONTRIBUTORS 31

APPENDIX C. REVISION HISTORY 32

COMMENTS AND FEEDBACK

In the spirit of open source, we invite anyone to provide feedback and comments on any reference architecture. Although we review our papers internally, sometimes issues or typographical errors are encountered. Feedback allows us to not only improve the quality of the papers we produce, but allows the reader to provide their thoughts on potential improvements and topic expansion to the papers. Feedback on the papers can be provided by emailing refarch-feedback@redhat.com. Please refer to the title within the email.

CHAPTER 1. EXECUTIVE SUMMARY

This reference architecture sets up two clusters, one is the **`Red Hat JBoss Data Grid (JDG) 7`** cluster and the other an **`Apache Spark 1.6`** cluster. Apache Spark processes data in memory, utilizing JDG 7 's in-memory data replication, eliminating the bottleneck that exists in many current enterprise applications.

This reference architecture also includes and deploys a sample **internet of things** (IoT) sensor application, which is developed based on a *quickstarts* example of JDG 7.

Since typical IoT workloads require low latency reads, writes and capacity to scale, JDG 7 cluster here serves not only as a highly scalable, high-performance data source for the Apache Spark cluster, but also ingests and serves data from IoT sensors. Spark tasks can operate on data stored in JDG caches with all the power of Spark batch and stream operators.

The goal is to provide a thorough description of the steps required for using Apache Spark with JDG 7 through the new Spark connector, and how to set up a cluster environment for JDG 7 and Spark.

CHAPTER 2. REFERENCE ARCHITECTURE ENVIRONMENT

2.1. APACHE SPARK

Apache Spark is a fast and general-purpose cluster computing system. It provides high-level APIs in **`Java`**, **`Scala`**, **`Python`** and **`R`**, and an optimized engine that supports general execution graphs. It also supports a rich set of higher-level tools including **`Spark SQL`** for **`SQL`** and structured data processing, **`MLlib`**for machine learning, **`GraphX`** for graph processing, and **`Spark Streaming`**.

2.2. RED HAT JBOSS DATA GRID 7

2.2.1. Overview

Red Hat's JBoss Data Grid 7 is an open source, distributed, in-memory key/value data store based on the Infinispan open source software project. Whether deployed in client/server mode or embedded in a Java Virtual Machine, it is built to be elastic, high performance, highly available and to scale linearly. Refer to the JDG 7 documentation for further details.

JBoss Data Grid is accessible for both Java and non-Java clients. Using JBoss Data Grid, data is distributed and replicated across a manageable cluster of nodes, optionally written to disk and easily accessible using the *REST*, *Memcached* and *Hot Rod* protocols, or directly in process through a traditional Java *Map* API.

2.2.2. JBoss Data Grid Usage Modes

Red Hat JBoss Data Grid offers two usage modes:

- Remote Client-Server mode
- Library mode

2.2.2.1. Library Mode

Library mode allows building and deploying a custom runtime environment. The Library mode hosts a single data grid node in the application process, with remote access to nodes hosted in other JVMs. Refer to the JDG 7 documentation for further details.

2.2.2.2. Remote Client-Server Mode

Remote Client-Server mode provides a managed, distributed, and clusterable data grid server. In Client-Server mode, the server runs as a self-contained process, utilizing a container based on **`Red Hat JBoss Enterprise Application Platform (JBoss EAP)`**, allowing client applications to remotely access the data grid server using **Hot Rod**, **Memcached** or **REST** client APIs. Refer to the JDG 7 documentation for further details.

2.2.3. Apache Spark Integration

The use of a dedicated JVM for JBoss Data Grid allows for appropriate tuning and configuration for JDG versus Spark and client applications. In particular, handling the memory requirements of both Apache Spark and JBoss Data Grid in a single JVM can be difficult. For this and other reasons, Apache Spark integration support is only provided in Client-Server mode and this reference

architecture sets up JBoss Data Grid accordingly.

JDG 7.0 introduces Resilient Distributed Dataset (RDD) and Discretized Stream (DStream) integration with Apache Spark version 1.6.0. This enables you to use JDG as a highly scalable, high-performance data source for Apache Spark, executing Spark and Spark Streaming operations on data stored in JDG. Refer to the JDG 7 documentation for further details.

CHAPTER 3. CREATING THE ENVIRONMENT

3.1. PREREQUISITES

Prerequisites for creating this reference architecture include a supported Operating System and JDK. Refer to Red Hat documentation for JBoss Data Grid 7.0 Supported Configurations.

3.2. DOWNLOADS

Download the attachments to this document. These application code and files will be used in configuring the reference architecture environment:

https://access.redhat.com/node/2640031/40/0

If you do not have access to the Red Hat customer portal, See the Comments and Feedback section to contact us for alternative methods of access to these files.

Download the Red Hat JBoss Data Grid 7.0.0 Server from Red Hat's Customer Support Portal

Download Apache Spark 1.6 from Apache Spark website download page

This reference architecture use the *spark-1.6.0-bin-hadoop2.6* build

3.3. INSTALLATION

3.3.1. Apache Spark

Installing Apache Spark is very simple and mainly involves extracting the downloaded archive file on each node.

```
# tar xvf spark-1.6.0-bin-hadoop2.6.tgz
```

3.3.2. JBoss Data Grid 7

JBoss Data Grid 7 does not require any installation steps. The archive file simply needs to be extracted after the download. This reference architecture requires installation of JBoss Data Grid 7.0.0 Server on each node.

```
# unzip jboss-datagrid-7.0.0-server.zip -d /opt/
```

3.4. CONFIGURATION

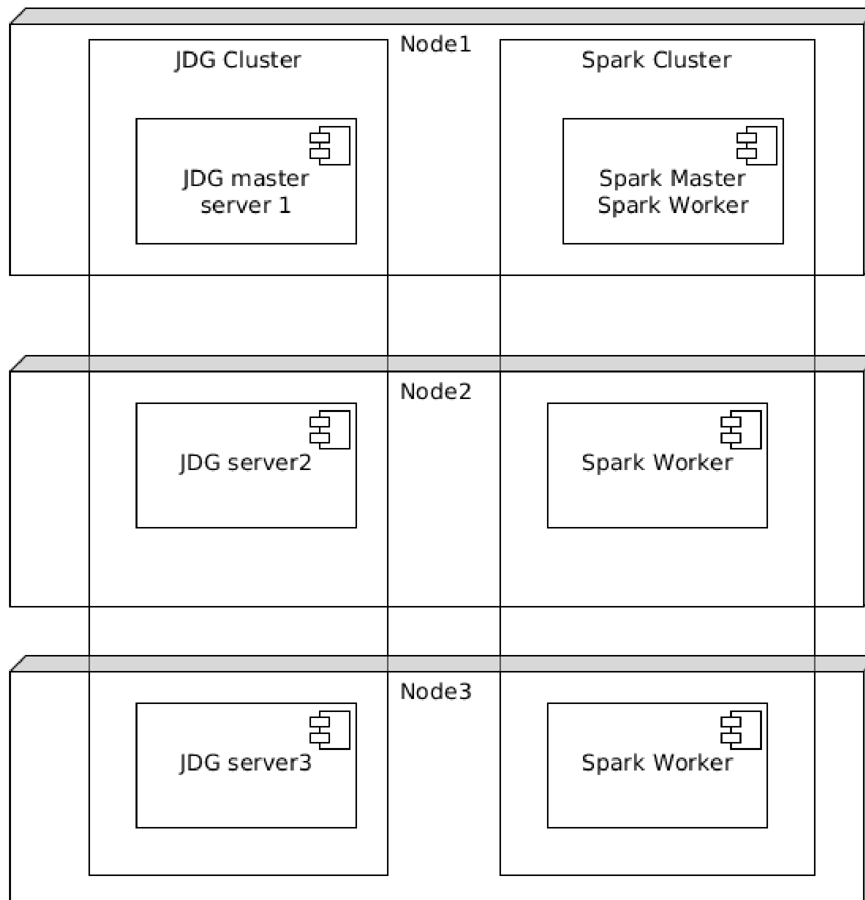
3.4.1. Overview

Various other types of configuration may be required for *UDP* and *TCP* communication. For example, Linux operating systems typically have a low maximum socket buffer size configured, which is lower than the default cluster *JGroups* buffer size. It may be important to correct any such warnings observed in the JDG logs. For more information, please follow the Administration and Configuration Guide for JDG 7

3.4.2. JDG 7 configuration

This reference architecture installs and configures a three-node cluster on separate machines. The names *node1*, *node2* and *node3* are used in this paper to refer to both the machines and the JDG 7 nodes on them.

Figure 3.1. Deployment Clusters



3.4.2.1. Adding Users

The first important step in configuring the JDG 7 clusters is to add the required users. They are Admin Users and Node Users.

1) Admin User

An administrator user is required for each domain. Assuming the user ID of *admin* and the password of *password1!* for this *admin* user:

On *node1*:

```
# /opt/jboss-datagrid-7.0.0-server/bin/add-user.sh admin password1!
```

This uses the non-interactive mode of the *add-user* script, to add management users with a given username and password.

2) Node Users

The next step is to add a user for each node that will connect to the cluster. That means creating two users called node2 and node3 (since node1 hosts the domain controller and does not need to use a password to authenticate against itself). This time, provide no argument to the add-user script and instead follow the interactive setup.

The first step is to specify that a management user is being added. The interactive process is as follows:

```
# /opt/jboss-datagrid-7.0.0-server/bin/add-user.sh
```

What type of user do you wish to add?
a) Management User (mgmt-users.properties)
b) Application User (application-users.properties)
(a): **a**

- Simply press enter to accept the default selection of *a*

Enter the details of the new user to add.
Realm (ManagementRealm) :

- Once again simply press enter to continue

Username : **node2**

- Enter the username and press enter (node1, node2 or node3)

Password : **password1!**

- Enter *password1!* as the password, and press enter

Re-enter Password : **password1!**

- Enter *password1!* again to confirm, and press enter

About to add user 'node_X_' for realm 'ManagementRealm'
Is this correct yes/no? **yes**

- Type *yes* and press enter

The continue:

Is this new user going to be used for one AS process to connect to another AS process?
e.g. for a slave host controller connecting to the master or for a Remoting connection for server to server EJB calls.
yes/no?

- Type *yes* and press enter

To represent the user add the following to the server-identities definition <secret value="cGFzc3dvcmQxIQ==" />

This concludes the setup of required management users to administer the domains and connect the

slave machines.

3.4.3. JDG 7 Cache configuration

On *node1*, execute the following scripts to add two new JDG 7 distributed caches, *sensor-data* and *sensor-avg-data*. These 2 caches will be used by the sample IoT sensor application.

```
# /opt/jboss-datagrid-7.0.0-server/bin/cli.sh # embed-host-controller -
-domain-config=domain.xml --host-config=host.xml --std-out=echo #
/profile=clustered/subsystem=datagrid-infinispan/cache-
container=clustered/configurations=CONFIGURATIONS/distributed-cache-
configuration=sensor-data:add(start=EAGER,template=false,mode=SYNC) #
/profile=clustered/subsystem=datagrid-infinispan/cache-
container=clustered/distributed-cache=sensor-
data:add(configuration=sensor-data) #
/profile=clustered/subsystem=datagrid-infinispan/cache-
container=clustered/configurations=CONFIGURATIONS/distributed-cache-
configuration=sensor-avg-data:add(start=EAGER,template=false,mode=SYNC)
# /profile=clustered/subsystem=datagrid-infinispan/cache-
container=clustered/distributed-cache=sensor-avg-
data:add(configuration=sensor-avg-data)
```

3.4.4. JDG 7 Cluster configuration

1) Update */opt/jboss-datagrid-7.0.0-server/domain/configuration/host-slave.xml* on both *node2* and *node3*, so these 2 nodes can form a JDG cluster with *node1*.

Update the first line for node2, adding host name.

<host name="node2" xmlns="urn:jboss:domain:4.0">

Update the first line for node3, adding host name.

<host name="node3" xmlns="urn:jboss:domain:4.0">

2) Update node2 and node3's *host-slave.xml*, change *server-identities* value from default sample value to this new value, which is the encrypted password value of the node user from last section.

<server-identities>
  <secret value="cGFzc3dvcmQxIQ=="/>
</server-identities>

3) Update the server name for node1 in *host.xml*, by deleting the *server-two* tag.

<server name="server-two" group="cluster" auto-start="true">

Update the server name to *server-two* for node2 in *host-slave.xml*.

<server name="server-two" group="cluster"/>

Update the server name to *server-three- for node3 in _host-slave.xml* .

<server name="server-three" group="cluster"/>

After these change, the cluster will have 3 members, *server-one* on node1, *server-two* on node2 and *server-three* on node3.

3.5. STARTUP

To start the active domain, assuming that *10.19.137.34* is the IP address for the *node1* machine, *10.19.137.35* for *node2* and *10.19.137.36* for *node3*:

3.5.1. Start JDG 7.0 cluster

Log on to the three machines where JDG 7 is installed and navigate to the *bin* directory:

```
# cd /opt/jboss-datagrid-7.0.0-server/bin
```

To start the first node

```
# ./domain.sh -bmanagement=10.19.137.34 -b=10.19.137.34
```

To start the second node

```
# ./domain.sh -b=10.19.137.35 -bprivate=10.19.137.35 --master-
address=10.19.137.34 --host-config=host-slave.xml
```

To start the third node

```
# ./domain.sh -b=10.19.137.36 -bprivate=10.19.137.36 --master-
address=10.19.137.34 --host-config=host-slave.xml
```

3.5.2. Stop JDG 7.0 cluster

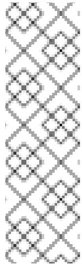
To stop the *sensor* applications, press *ctrl-c* or use *"kill -9 PID"* to stop the process.

3.5.3. Start Apache Spark cluster

Apache Spark currently supports three types of cluster managers:

- *Standalone* – a simple cluster manager included with Spark that makes it easy to set up a cluster.
- *Apache Mesos* – a general cluster manager that can schedule short-lived tasks and long-running services on shared compute resources.
- *Hadoop YARN* – the resource manager in Hadoop 2.

This reference architecture uses the *standalone* cluster mode.



Note

Each streaming receiver will use a CPU core / thread from the processors allocated to Apache Spark. Ensure that the Spark application always has a higher number of CPU cores than receivers. Failure to allocate at least one extra processing core can result in receivers running but no data being processed by Spark.

3.5.3.1. Start Apache Spark standalone cluster

By default, Apache Spark uses port 8080 for its Web UI, which is coincidentally the same port used by JBoss Data Grid, as configured in its *domain.xml*:

```
<socket-binding name="rest" port="8080"/>
```

Therefore, an attempt to start Apache Spark on the same host as JDG 7 may result in the following exception due to a port conflict:

```
ERROR [org.jboss.msc.service.fail] (MSC service thread 1-3) MSC000001: Failed to start service jboss.datagrid-infinispan-endpoint.rest.rest-connector: org.jboss.msc.service.StartException in service jboss.datagrid-infinispan-endpoint.rest.rest-connector: DGENDPT10016: Could not start the web context for the REST Server [Server:server-one] at org.infinispan.server.endpoint.subsystem.RestService.start(RestService.java:110)
```

To avoid such a port conflict, please start Apache Spark with the *--webui-port* argument to use a different port.

On Node 1 (10.19.137.34), start both master and worker.

```
# cd /opt/spark-1.6.0-bin-hadoop2.6/sbin # ./start-master.sh --webui-port 9080 -h 10.19.137.34 # ./start-slave.sh spark://10.19.137.34:7077 --webui-port 9081
```

On Node 2 (10.19.137.35), start one worker.

```
# cd /opt/spark-1.6.0-bin-hadoop2.6/sbin # ./start-slave.sh spark://10.19.137.34:7077 --webui-port 9081
```

On Node 3 (10.19.137.36), start one worker.

```
# cd /opt/spark-1.6.0-bin-hadoop2.6/sbin # ./start-slave.sh spark://10.19.137.34:7077 --webui-port 9081
```

3.5.4. Stop Apache Spark cluster

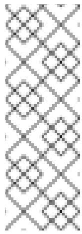
On Node1, stop both master and worker.

```
# cd /opt/spark-1.6.0-bin-hadoop2.6/sbin # ./stop-slave.sh # ./stop-master.sh
```

On Node2 and Node3, only need to stop the worker.

```
# cd /opt/spark-1.6.0-bin-hadoop2.6/sbin # ./stop-slave.sh
```

–



Note

The whole Apache Spark cluster can also be started and stopped using launch scripts, like sbin/start-all.sh and sbin/stop-all.sh, which need additional configuration. For details, please refer to Cluster Launch Scripts

3.5.5. Start IoT sensor application

3.5.5.1. Start Spark analysis application

```
 # /opt/spark-1.6.0-bin-hadoop2.6/bin/spark-submit --master
spark://10.19.137.34:7077 --deploy-mode cluster --supervise --class
org.Analyzer target/ref-analysis-jar-with-dependencies.jar
10.19.137.34:11222;10.19.137.35:11222;10.19.137.36:11222
```

The arguments provided to *spark-submit* are as follows:

- master: The master URL for the cluster
- deploy-mode cluster: deploy the driver on the worker nodes (**`cluster`**)
- supervise: to make sure that the driver is automatically restarted if it fails with non-zero exit code.
- class: entry point of the application
- The last argument is the JDG 7 cluster address, which is used in the Spark connector.

For more information on how to use spark-submit, please refer to this link

3.5.5.2. Start Client application

```
# java -jar target/temperature-client-jar-with-dependencies.jar
10.19.137.34 shipment1 shipment5 shipment9
```

The arguments to this application include:

- The first argument is the address for the *Hot Rod* Java Client to connect to JDG 7, in this example it is 10.19.137.34. Since the JDG cluster has three nodes (10.19.137.34, 10.19.137.35 and 10.19.137.36), using either one of the IP addresses will work.
- After that, it's the shipment ID strings that the client will be listening to. It doesn't have to be an exact match, for example "shipment1" will bring back all shipments with an ID starting with "shipment1", like shipment101 or shipment18.

3.5.5.3. Start Sensor application

```
# java -jar target/temperature-sensor-jar-with-dependencies.jar
10.19.137.34
```

The first argument is the address for the *Hot Rod* Java Client to connect to JDG 7, in this example it is 10.19.137.34. Since the JDG cluster has three nodes (10.19.137.34, 10.19.137.35 and 10.19.137.36), using either one of the IP addresses is fine.

3.5.6. Stop IoT sensor application

To stop the Sensor applications, press *ctrl-c* or use *"kill -9 PID"* to stop the process. Otherwise, all 3 applications are set up to run for 24 hours.

CHAPTER 4. SENSOR APPLICATION

4.1. OVERVIEW

A global transportation company can have thousands of IoT sensors installed in various places like gates, inside containers, ships, and planes. These sensors continuously feed a large amount of data to the data center. Processing the large amount of data provided by this feed in a reasonable amount of time is the biggest challenge of the project. Apache Spark's lighting fast processing speed combines with the high performance, in-memory, distributed, NoSQL datastore solution provided by JBoss Data Grid 7 to fulfill this requirement. The second challenge is related to the dynamic model of a global logistics business, where certain periods, like the time before major holidays, might need more processing power than others. This requires processing capability to be added on-the-fly. Again, Apache Spark's large-scale data processing with the elastic scalability of JDG 7 meet this requirement.

This reference architecture includes and deploys a sample IoT sensor application, which makes use of the *Spark connector*, providing tight integration with Apache Spark, and allowing applications written in *Scala* to utilize JBoss Data Grid as a backing data store. This connector includes support for the following:

Spark connector includes support for the following:

- Create an RDD from any cache
- Write a key/value RDD to a cache
- Create a DStream from cache-level events
- Write a key/value DStream to a cache

4.2. PACKAGES OVERVIEW

This application contains three main packages, and a common data package (*ShipmentData*) shared by all three packages.

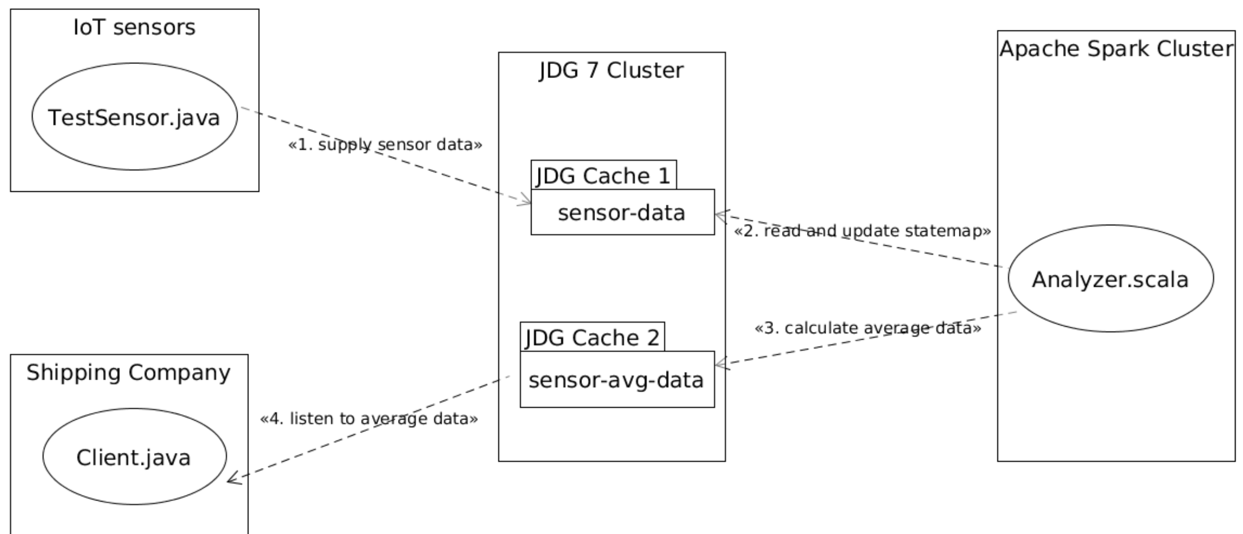
The first package is *Sensor* with its main class being *TestSensor.java*, which simulates IoT sensors for a global transportation company. These sensors might be installed anywhere, like on docks, ships, gates, containers, trucks, warehouses, etc. These sensors send temperature and humidity data they collect to the JDG cluster's *sensor-data* cache.

The second package is *Analyzer* with the main class *Analyzer.scala*, which runs on the Apache Spark cluster, reads the sensor-data cache through the JDG Spark connector, and calculates the average temperature and humidity in the past 5 minutes, putting the data in the JDG *sensor-avg-data* cache.

The third package is *Client* with the main class *Client.java*, which simulates system clients; the client might be a transportation company or owners of the actual shipment, who are interested in the condition of the shipment. They will monitor the *sensor-avg-data* cache and send alerts if there might be a problem.

Below is the data flow diagram.

Figure 4.1. Data Flow Diagram



Last is the *ShipmentData.java* class, which is a simple data object:

```
@SuppressWarnings("serial")
public class ShipmentData implements Serializable {
 private double temperature;
 private double humidity;
 private String skuNumbers;
 private long sensorTime;


 public void setSensorTime(long sensorTime) {
  this.sensorTime = sensorTime;
 }

 public double getTemperature() {
  return temperature;
 }
 public void setTemperature(double temperature) {
  this.temperature = temperature;
 }
 public double getHumidity() {
  return humidity;
 }
 public void setHumidity(double humidity) {
  this.humidity = humidity;
 }
 public String getSkuNumbers() {
  return skuNumbers;
 }
 public void setSkuNumbers(String skuNumbers) {
  this.skuNumbers = skuNumbers;
 }
 public long getSensorTime() {
  return sensorTime;
 }

 public String getValues(){
  SimpleDateFormat sdf = new SimpleDateFormat("MMM dd,yyyy HH:mm");
  Date sensorTimeInDisplay = new Date(sensorTime);
  return " timeGenerated: " + sdf.format(sensorTimeInDisplay) + "
```

```
temperature: " + (int)this.temperature + " humidity: " +
(int)this.humidity + " skuNumbers " + this.skuNumbers;
 }
}
```

4.3. SENSOR PACKAGE

The *TestSensor* class sends *ShipmentData* to the JDG *sensor-data* cache, simulating the real world work of an IoT sensor. The key for the cache is the shipment ID, like shipment1, shipment2,… shipment1000, while the value is the *ShipmentData* object.

```
public class TestSensor {

 public static final String CACHE_NAME = "sensor-data";
 private static final Random RANDOM = new Random();

 // running for 24 hours
 private static final int GENERATE_INTERVAL = 1000 * 60 * 60 * 24;
 // how often data should be generated, in ms
 private static final int GENERATE_PERIOD = 1000;

 public static void main(String[] args) throws InterruptedException {

  // Configure remote cache
  ConfigurationBuilder builder = new ConfigurationBuilder();
  String DATAGRID_IP = args[0];

builder.addServer().host(DATAGRID_IP).port(ConfigurationProperties.DEFA
ULT_HOTROD_PORT);
  RemoteCacheManager cacheManager = new
RemoteCacheManager(builder.build());
  RemoteCache<String, ShipmentData> cache =
cacheManager.getCache(CACHE_NAME);

  // Insert some sensor data into the cache
  TimerTask randomSensorData = new SensorDataGenerator(cache);
  Timer timer = new Timer(true);
  timer.schedule(randomSensorData, 0, GENERATE_PERIOD);

  // Generate sensor data for specified interval
  Thread.sleep(GENERATE_INTERVAL);
  // Thread.sleep(GENERATE_INTERVAL * 60 * 1000);
  randomSensorData.cancel();
  cacheManager.stop();
  System.exit(0);
 }

 private static class SensorDataGenerator extends TimerTask {
  // maximum temperature for the simulation data range
  private static final int TEMP_MAX = 60;
  // maximum humidity for the simulation data range
  private static final int HUMIDITY_MAX = 100;

  private final RemoteCache<String, ShipmentData> cache;
```

```
    public SensorDataGenerator(RemoteCache<String, ShipmentData> cache) {
     this.cache = cache;
    }

    @Override
    public void run() {
     //Simulate 1000 different shipments
     int i = RANDOM.nextInt(1000);

     String shipment = "shipment" + i;
     String sku = "sku_";
     double temperature = RANDOM.nextDouble() * TEMP_MAX;
     double humidity = RANDOM.nextDouble() * HUMIDITY_MAX;
     // setup the fake data, that the first batch of sku# is for meat
shipment, next batch is metal, next batch is animal, last batch for
combined shipment
     if (i < 250) {
      sku = sku + "meat" + i;
      temperature = RANDOM.nextDouble() * 30;
     }
     if (i >= 250 && i < 500) {
      sku = sku + "metal" + i;
      humidity = RANDOM.nextDouble() * 40;
     }
     if (i >= 500 && i < 750) {
      sku = sku + "animal" + i;
      temperature = RANDOM.nextDouble() * 45;
      humidity = RANDOM.nextDouble() * 75;
     }
     if (i >= 750 && i < 1000) {
      sku = sku + "animal" + i + "," + sku + "metal" + i;
     }

     ShipmentData newShipment = new ShipmentData();
     newShipment.setHumidity(humidity);
     newShipment.setTemperature(temperature);
     newShipment.setSkuNumbers(sku);
     newShipment.setSensorTime(System.currentTimeMillis());

     cache.put(shipment, newShipment);
     System.out.println("Inserted " + shipment +
newShipment.getValues());
    }
   }

}
```

4.3.1. TestSensor class detail explanation

SensorDataGenerator is the inner class which extends *TimerTask*; the main purpose for this class is to generate the data which simulates real world *IoT* data:

```
 private static class SensorDataGenerator extends TimerTask {
```

To simulate the data of IoT sensors from all over the world, this class will generate temperature and humidity data, and set these values for random shipments (shipment id from 0 to 999).

The key for ShipmentData is the *shipment* field; the value of it won't change during the whole trip, just like in the real world. Each shipment could contain different items, which are identified by SKU numbers; it could be one or multiple items, and during the trip, the SKU number won't change as well.

For demonstration purposes and in our example, if the random shipment id is between 0 to 250, the SKU number will be assigned as meat shipment. If the random shipment id is between 250 to 500, the SKU number will be assigned as metal shipment. if the random shipment id is between 500 to 750, the SKU number will be assigned as animal shipment. For shipment id between 750 to 1000, it's a concatenated string of metal and animal, to demo a shipment that has both animal and metal products in it.

```
   String shipment = "shipment" + i;
   String sku = "sku_";
   double temperature = RANDOM.nextDouble() * TEMP_MAX;
   double humidity = RANDOM.nextDouble() * HUMIDITY_MAX;
   // setup the fake data, that the first batch of sku# is for meat
shipment, next batch is metal, next batch is animal, last batch for
combined shipment
   if (i < 250) {
    sku = sku + "meat" + i;
    temperature = RANDOM.nextDouble() * 30;
   }
   if (i >= 250 & i < 500) {
    sku = sku + "metal" + i;
    humidity = RANDOM.nextDouble() * 40;
   }
   if (i >= 500 & i < 750) {
    sku = sku + "animal" + i;
    temperature = RANDOM.nextDouble() * 45;
    humidity = RANDOM.nextDouble() * 75;
   }
   if (i >= 750 & i < 1000) {
    sku = sku + "animal" + i + "," + sku + "metal" + i;
   }
```

The *main* method of the *TestSensor* class will set up the Hot Rod Java Client to JDG cluster and connect to the *sensor-data* cache:

```
  // Configure remote cache
  ConfigurationBuilder builder = new ConfigurationBuilder();
  String DATAGRID_IP = args[0];

builder.addServer().host(DATAGRID_IP).port(ConfigurationProperties.DEFA
ULT_HOTROD_PORT);
  RemoteCacheManager cacheManager = new
RemoteCacheManager(builder.build());
  RemoteCache<String, ShipmentData> cache =
cacheManager.getCache(CACHE_NAME);
```

Then it will start the timer and assign the task, which generates random sensor data.

```
    // Insert some sensor data into the cache
    TimerTask randomSensorData = new SensorDataGenerator(cache);
    Timer timer = new Timer(true);
    timer.schedule(randomSensorData, 0, GENERATE_PERIOD);
```

e.g. Below is sample of random data output.

Inserted shipment546 timeGenerated: Aug 22,2016 14:42 temperature: 30 humidity: 35 skuNumbers sku_animal546
Inserted shipment333 timeGenerated: Aug 22,2016 14:42 temperature: 11 humidity: 6 skuNumbers sku_metal333
Inserted shipment571 timeGenerated: Aug 22,2016 14:42 temperature: 33 humidity: 13 skuNumbers sku_animal571
Inserted shipment942 timeGenerated: Aug 22,2016 14:42 temperature: 47 humidity: 3 skuNumbers sku_animal942,sku_metal942

4.4. ANALYZER PACKAGE

The *Analyzer* class reads shipment data from the JDG 7 *sensor-data* cache based on the *shipmentID* key, then calculates average temperature and humidity for the past 5 minutes for each shipment, and stores the average *ShipmentData* to another (*sensor-avg-data*) JDG 7 cache, also storing the latest shipment data back in the map in the *sensor-data* cache.

```
/**
 * <p>Computes average sensor data in each place from incoming stream
of measurements.</p>
 * <p>The measurement stream is read from the JDG's sensor-data and
produces stream of average data.
 * Updates are written back to the Data Grid into a different cache
called sensor-avg-data.
 */
object Analyzer {

  val inputDataGridCache = "sensor-data"
  val outputDataGridCache = "sensor-avg-data"



  val mapFunc = (place: String, currDataLists:
Option[Iterable[ShipmentData]], state: State[Map[String,
Iterable[ShipmentData]]]) => {

    // obtain the state map contains the past 5 minutes shipment data
    val stateMap: Map[String, Iterable[ShipmentData]] =
state.getOption().getOrElse(Map[String, Iterable[ShipmentData]]())
    val pastData: (Iterable[ShipmentData]) = stateMap.getOrElse(place,
(Iterable[ShipmentData])())

    var sumTemp: Double = 0d;
    var sumHumd: Double = 0d;
    val currTime: Long = java.lang.System.currentTimeMillis();
    var skuNos: String = "";

    //create the new list for storing the data, from both pastData and
```

```
currData
    var newShipmentData = collection.mutable.ListBuffer[ShipmentData]
();

    //only count the time in less than 5 minutes
    pastData.foreach { data: ShipmentData =>
      val sensorTime: Long = data.getSensorTime();

      //for testing usage
      val sensorTimeInDisplay = new Date(sensorTime);

      //only include the data from past 5 minutes
      if (currTime - sensorTime < 5 * 60 * 1000) {
        sumTemp = sumTemp + data.getTemperature();
        sumHumd = sumHumd + data.getHumidity();
        newShipmentData += data;
      }
    }

    val currData: Iterable[ShipmentData] =
currDataLists.getOrElse(List());

    currData.foreach { data: ShipmentData =>
      //only count the time in less than 5 minutes
      val sensorTime: Long = data.getSensorTime();

      //for testing usage
      val sensorTimeInDisplay = new Date(sensorTime);
      //only include the data from past 5 minutes
      if (currTime - sensorTime < 5 * 60 * 1000) {
        sumTemp = sumTemp + data.getTemperature();
        sumHumd = sumHumd + data.getHumidity();
        newShipmentData += data;
        skuNos = data.getSkuNumbers();
      }
    }

    val count = newShipmentData.size;
     var avgTemp: Double = sumTemp / count;
    var avgHumd: Double = sumHumd / count;

    var avgShipmentData: ShipmentData = new ShipmentData();
    avgShipmentData.setTemperature(avgTemp);
    avgShipmentData.setHumidity(avgHumd);
    avgShipmentData.setSkuNumbers(skuNos);
    avgShipmentData.setSensorTime(currTime);

    // update stored state
    state.update(stateMap.updated(place, newShipmentData.toList))

    (place, avgShipmentData)
  }


  def main(args: Array[String]) {
    val usage = """
```

```
    Please input the Spark server address in ip:port format
  """

    if (args.length == 0) println(usage)
    val dataGridServer = args(0)
    // Initialize the Spark streaming context, with a batch duration of
1 second.
    val sparkConf = new SparkConf().setAppName("SparkSensor")
    val ssc = new StreamingContext(sparkConf, Seconds(1))
    // set up checkpoint to store state information
    ssc.checkpoint("/mnt/shared/sharefs/spark-sensor")

    // configure the connection to the DataGrid and create the incoming
DStream
    val configIn = new Properties
    configIn.put("infinispan.rdd.cacheName", inputDataGridCache)
    configIn.put("infinispan.client.hotrod.server_list",
dataGridServer)
    val ispnStream = new InfinispanInputDStream[String, ShipmentData]
(ssc, StorageLevel.MEMORY_ONLY, configIn)

    // extract the (place, temperature) pair from the incoming stream
that is composed of (key, value, eventType)
    // we are assuming only events of type CLIENT_CACHE_ENTRY_CREATED
will be present.
    val measurementStream: DStream[(String, ShipmentData)] = ispnStream
map { case (key, value, eventType) => (key, value) }

    // as measurements are batched, it can contain several measurements
from the same place - let's group them together
    val measurementGrouped: DStream[(String, Iterable[ShipmentData])] =
measurementStream.groupByKey()

    // Produce a new DStream combining the previous DStream with the
stored state
    val avgSensorDataStream: MapWithStateDStream[String,
Iterable[ShipmentData], Map[String, Iterable[ShipmentData]], (String,
ShipmentData)] =
      measurementGrouped.mapWithState(StateSpec.function(mapFunc))

    // just print number of items in each RDD
    avgSensorDataStream.foreachRDD(rdd => {
      printf("# items in DStream: %d\n", rdd.count())
      rdd.foreach { case (place, average) => println("Averages:" +
place + " -> " + average) }
    })

    // write stream to the avg-sensor-data cache
    val configOut = new Properties
    configOut.put("infinispan.rdd.cacheName", outputDataGridCache)
    configOut.put("infinispan.client.hotrod.server_list",
dataGridServer)
    avgSensorDataStream.writeToInfinispan(configOut)
```

```
    ssc.start()
    ssc.awaitTermination()
  }
}
```

4.4.1. Analyzer class detail explanation

First declare both JDG caches that will be used in this class:

```
  val inputDataGridCache = "sensor-data"
  val outputDataGridCache = "sensor-avg-data"
```

Get the JDG 7 server address from the first argument provided when starting the server, it can be the full cluster address like this: *10.19.137.34:11222;10.19.137.35:11222;10.19.137.36:11222*

```
    val dataGridServer = args(0)
```

Next, initialize settings for spark stream and checkpoint path.

```
    // Initialize the Spark streaming context, with a batch duration of
1 second.
    val sparkConf = new SparkConf().setAppName("SparkSensor")
    val ssc = new StreamingContext(sparkConf, Seconds(1))
    // set up checkpoint to store state information
    ssc.checkpoint("/mnt/shared/sharefs/spark-sensor")
```

Then create the Spark stream.

```
    // configure the connection to the DataGrid and create the incoming
DStream
    val configIn = new Properties
    configIn.put("infinispan.rdd.cacheName", inputDataGridCache)
    configIn.put("infinispan.client.hotrod.server_list",
dataGridServer)
    val ispnStream = new InfinispanInputDStream[String, ShipmentData]
(ssc, StorageLevel.MEMORY_ONLY, configIn)
```

Next section is the kernel of this class, it extracts data from incoming stream and call the mapFunc function to calculate the average sensor data.

```
    // extract the (place, temperature) pair from the incoming stream
that is composed of (key, value, eventType)
    // we are assuming only events of type CLIENT_CACHE_ENTRY_CREATED
will be present.
    val measurementStream: DStream[(String, ShipmentData)] = ispnStream
map { case (key, value, eventType) => (key, value) }

    // as measurements are batched, it can contain several measurements
from the same place - let's group them together
    val measurementGrouped: DStream[(String, Iterable[ShipmentData])] =
measurementStream.groupByKey()

    // Produce a new DStream combining the previous DStream with the
stored state
```

```
    val avgSensorDataStream: MapWithStateDStream[String,
Iterable[ShipmentData], Map[String, Iterable[ShipmentData]], (String,
ShipmentData)] =
      measurementGrouped.mapWithState(StateSpec.function(mapFunc))
```

Last part of the *main* method will just write to the JDG 7 cache (*avg-sensor-data*).

```
    // write stream to the avg-sensor-data cache
    val configOut = new Properties
    configOut.put("infinispan.rdd.cacheName", outputDataGridCache)
    configOut.put("infinispan.client.hotrod.server_list",
dataGridServer)
    avgSensorDataStream.writeToInfinispan(configOut)
```

The other funciton (*mapFunc*) in this class focuses on the getting the past 5-minute sensor data and calculating averages based on it.

First, get the existing *statemap*, and initialize other variables like the new average shipment data collection.

```
    // obtain the state map contains the past 5 minutes shipment data
    val stateMap: Map[String, Iterable[ShipmentData]] =
state.getOption().getOrElse(Map[String, Iterable[ShipmentData]]())
    val pastData: (Iterable[ShipmentData]) = stateMap.getOrElse(place,
(Iterable[ShipmentData])())

    var sumTemp: Double = 0d;
    var sumHumd: Double = 0d;
    val currTime: Long = java.lang.System.currentTimeMillis();
    var skuNos: String = "";

    //create the new list for storing the data, from both pastData and
currData
    var newShipmentData = collection.mutable.ListBuffer[ShipmentData]
();
```

Then get the total sum from both history data and current data.

```
    //only count the time in less than 5 minutes
    pastData.foreach { data: ShipmentData =>
      val sensorTime: Long = data.getSensorTime();

      //for testing usage
      val sensorTimeInDisplay = new Date(sensorTime);

      //only include the data from past 5 minutes
      if (currTime - sensorTime < 5 * 60 * 1000) {
        sumTemp = sumTemp + data.getTemperature();
        sumHumd = sumHumd + data.getHumidity();
        newShipmentData += data;
      }
    }

    val currData: Iterable[ShipmentData] =
currDataLists.getOrElse(List());
```

```
      currData.foreach { data: ShipmentData =>
        //only count the time in less than 5 minutes
        val sensorTime: Long = data.getSensorTime();

        //for testing usage
        val sensorTimeInDisplay = new Date(sensorTime);
        //only include the data from past 5 minutes
        if (currTime - sensorTime < 5 * 60 * 1000) {
          sumTemp = sumTemp + data.getTemperature();
          sumHumd = sumHumd + data.getHumidity();
          newShipmentData += data;
          skuNos = data.getSkuNumbers();
        }
      }
```

Last part calculates the average data and updates *statemap*.

```
      val count = newShipmentData.size;
       var avgTemp: Double = sumTemp / count;
      var avgHumd: Double = sumHumd / count;

      var avgShipmentData: ShipmentData = new ShipmentData();
      avgShipmentData.setTemperature(avgTemp);
      avgShipmentData.setHumidity(avgHumd);
      avgShipmentData.setSkuNumbers(skuNos);
      avgShipmentData.setSensorTime(currTime);

      // update stored state
      state.update(stateMap.updated(place, newShipmentData.toList))

      (place, avgShipmentData)
```

4.5. CLIENT PACKAGE

The *Client* class simulates the real world end user, like the transportation company watching closely for its shipment. It contains the business rules based on SKU number, if the shipment data reaches the warning level, etc.

It registers a listener on the JDG *sensor-avg-data* cache for the shipment and when the *shipmentID* matches, it will read the average temperature and humidity data of the past 5 minutes and compare them with the threshold. Inside the *ShipmentData* class there is a field called *skuNumbers*, which contains one or more SKU numbers. *Client.java* uses the *skuNumbers* to get the related threshold rules and compare with the current average temperature and humidity to decide whether to send out a warning or not.

Examples: 1) Sensor sends *shipment1* to 1st JDG cache

Inserted shipment1 timeGenerated: Aug 22,2016 15:07 temperature: 19 humidity: 67 skuNumbers sku_meat1
Inserted shipment1 timeGenerated: Aug 22,2016 15:07 temperature: 17 humidity: 61 skuNumbers sku_meat1

2) Spark Analyzer reads from 1st JDG cache, calcualtes the *shipment1* average temperature value as 18 and stores it in the 2nd JDG cache.

3) Client is listening on the 2nd JDG cache for *shipment1*, so it reads the average temperature in the past 5 minutes as 18, and also reads the skuNumbers as *sku_meat1*. Since for meat shipments, there is a threshold that temperature can't be higher than 15 degrees, it sends out the below warning:

Average data is from: shipment1 timeGenerated: Aug 22,2016 15:07 temperature: 18 humidity: 64 skuNumbers sku_meat1
Warning, MEAT shipment is in danger, current temperature: 18 threshold is 15

This package contains two classes. First class is *Client.java*, the main purpose of which is to create the Hot Rod Java Client connection and add the listener to the 2nd JDG cache (*sensor-avg-data*).

```
public class Client {

   public static final String CACHE_NAME = "sensor-avg-data";

   public static void main(String[] args) throws Exception {
      // check provided arguments - at least one shipment needs to be
specified
      if (args.length < 1) {
         System.err.println("You have to provide list of shipment to
watch, at least one!");
         System.exit(1);
      }
      String DATAGRID_IP = args[0];
      args[0] = null;
      Set<String> shipmentToWatch = new HashSet<>(args.length);
      Collections.addAll(shipmentToWatch, args);

      // Configure remote cache
      ConfigurationBuilder builder = new ConfigurationBuilder();

builder.addServer().host(DATAGRID_IP).port(ConfigurationProperties.DEFA
ULT_HOTROD_PORT);
      RemoteCacheManager cacheManager = new
RemoteCacheManager(builder.build());
      RemoteCache<String, ShipmentData> cache =
cacheManager.getCache(CACHE_NAME);

      // Add cache listener and wait for specified amount of time
      AvgSensorDataListener avgListener = new
AvgSensorDataListener(cache, shipmentToWatch);
      cache.addClientListener(avgListener);
      int LISTEN_TIME = 24; // how long the client should listen to
changes, in hours
      System.out.printf("Client will be listening to avg. sensor
updates for %d hours%n", LISTEN_TIME);
      Thread.sleep(LISTEN_TIME * 60 * 60 * 1000);

      System.out.println("Stopping client");
      cache.removeClientListener(avgListener);
      cacheManager.stop();
      System.exit(0);
   }

}
```

-

4.5.1. Client class detail explanation

The first part of Client's *main* method is the java argument handling, which deals with the IP addresses and shipment ids that it needs to listen to.

```
      // check provided arguments - at least one shipment needs to be
specified
      if (args.length < 1) {
         System.err.println("You have to provide list of shipment to
watch, at least one!");
         System.exit(1);
      }
      String DATAGRID_IP = args[0];
```

It corrosponds to the invoke command of *Client.java*

```
# java -jar target/temperature-client-jar-with-dependencies.jar
10.19.137.34 shipment1 shipment5 shipment9
```

Those shipment id strings that the client is interested in, will be put into a collection, then passed to the second class, *AvgSensorDataListener*.

```
      Set<String> shipmentToWatch = new HashSet<>(args.length);
      Collections.addAll(shipmentToWatch, args);
```

The next part will create the Hot Rod Java Client to JDG cluster and connect with the *sensor-avg-data* JDG cache.

```
// Configure remote cache
      ConfigurationBuilder builder = new ConfigurationBuilder();

builder.addServer().host(DATAGRID_IP).port(ConfigurationProperties.DEFA
ULT_HOTROD_PORT);
      RemoteCacheManager cacheManager = new
RemoteCacheManager(builder.build());
      RemoteCache<String, ShipmentData> cache =
cacheManager.getCache(CACHE_NAME);
```

Then it just registers the listener:

```
// Add cache listener and wait for specified amount of time
      AvgSensorDataListener avgListener = new
AvgSensorDataListener(cache, shipmentToWatch);
      cache.addClientListener(avgListener);
      int LISTEN_TIME = 24; // how long the client should listen to
changes, in hours
      System.out.printf("Client will be listening to avg. sensor
updates for %d hours%n", LISTEN_TIME);
      Thread.sleep(LISTEN_TIME * 60 * 60 * 1000);
```

The second class in Client package is *AvgSensorDataListener.java*

```
@ClientListener
```

```
public class AvgSensorDataListener {
 private final RemoteCache<String, ShipmentData> cache;
 private final Set<String> watchedShipment;

 public AvgSensorDataListener(RemoteCache<String, ShipmentData> cache,
Set<String> watchedShipment) {
  this.cache = cache;
  this.watchedShipment = watchedShipment;
 }

 @ClientCacheEntryCreated
 public void entryCreated(ClientCacheEntryCreatedEvent<String> event) {
  for (String oneWatchedShipment : watchedShipment){
   if (event.getKey()!=null && oneWatchedShipment != null &&
event.getKey().contains(oneWatchedShipment))
    judgeAction(event.getKey());
  }

 }

 @ClientCacheEntryModified
 public void entryModified(ClientCacheEntryModifiedEvent<String> event)
{
  for (String oneWatchedShipment : watchedShipment){
   if (event.getKey()!=null && oneWatchedShipment != null &&
event.getKey().contains(oneWatchedShipment))
    judgeAction(event.getKey());
  }
 }

 private void judgeAction(String key) {
  ShipmentData data = cache.get(key);
  // setup a few test business rules, rules are based on SKU #.
  String skuNo = data.getSkuNumbers();
  if (skuNo.contains("meat") && data.getTemperature() > 15) {
   // business logic: meat's average temperature can't be higher than
15
   System.out.println("Average data is from: " + key +
data.getValues());
   System.out.println(" Warning, MEAT shipment is in danger,  current
temperature: " + (int) data.getTemperature() + " threshold is 15");
   System.out.println("");
  }
  if (skuNo.contains("metal") && data.getHumidity() > 30) {
   // business logic:metal's average humidity can't be higher than 30
   System.out.println("Average data is from: " + key +
data.getValues());
   System.out.println("  Warning, METAL shipment is in danger,  current
humidity: " + (int) data.getHumidity() + " threshold is 30");
   System.out.println("");
  }
  if (skuNo.contains("animal") && (data.getTemperature() < 15 ||
data.getTemperature() > 30 || data.getHumidity() < 20 ||
data.getHumidity()> 90)) {
   // business logic: animal average temperature can't be lower than 15
or higher than 30 and average humidity need to between 20 and 90
```

```
   System.out.println("Average data is from: " + key +
data.getValues());
   System.out.println(" Warning, ANIMAL shipment is in danger, current
temperature:  " + (int) data.getTemperature() + " threshold is between
15 to 30" + " current humidity: " + (int) data.getHumidity() + "
threshold is between 20 to 90");
   System.out.println("");
  }

 }
}
```

4.5.2. AvgSensorDataListener class detail explanation

Both *entryCreated* and *entryModified* methods will compare the watched shipment string with the current created/modified shipment string. If they are a match, then it invokes another method to see if the data will trigger the alert.

```
 @ClientCacheEntryCreated
 public void entryCreated(ClientCacheEntryCreatedEvent<String> event) {
  for (String oneWatchedShipment : watchedShipment){
   if (event.getKey()!=null && oneWatchedShipment != null &&
event.getKey().contains(oneWatchedShipment))
    updateAction(event.getKey());
  }

 }

 @ClientCacheEntryModified
 public void entryModified(ClientCacheEntryModifiedEvent<String> event)
{
  for (String oneWatchedShipment : watchedShipment){
   if (event.getKey()!=null && oneWatchedShipment != null &&
event.getKey().contains(oneWatchedShipment))
    updateAction(event.getKey());
  }
 }
```

When the right sensor data is encountered, the *judgeAction* method will be invoked. It uses the SKU number to decide whether the threshold is reached or not, to then send out an alert. Below is the code for meat shipment; the other cases are similar.

```
  ShipmentData data = cache.get(key);
  // setup a few test business rules, rules are based on SKU #.
  String skuNo = data.getSkuNumbers();
  if (skuNo.contains("meat") && data.getTemperature() > 15) {
   // business logic: meat's average temperature can't be higher than
15
   System.out.println("Average data is from: " + key +
data.getValues());
   System.out.println(" Warning, MEAT shipment is in danger,  current
temperature: " + (int) data.getTemperature() + " threshold is 15");
   System.out.println("");
  }
```

APPENDIX A. REVISION HISTORY

Revision	Release Date	Author(s)
1.0	Sep 2016	Calvin Zhu

APPENDIX B. CONTRIBUTORS

We would like to thank the following individuals for their time and patience as we collaborated on this process. This document would not have been possible without their many contributions.

Contributor	Title	Contribution
Babak Mozaffari	Manager, Software Engineering & Consulting Engineer	Technical Content Review
Matthew Farrellee	Senior Principal Software Engineer	Technical Content Review
William Benton	Principal Software Engineer	Technical Content Review
Emmanuel Bernard	Consulting Software Engineer	Technical Content Review
Gustavo Fernandes	Principal Software Engineer	Technical Content Review

APPENDIX C. REVISION HISTORY

Revision 1.0-0	Sep 2016	CZ