Build and Test instructions of BigData Apache components

- About BigData components
- Build and Test Instructions
  - Install OpenJDK 8
  - Install Maven v3.3.9
- Flink
- Kafka
- Oozie
- NiFi
- Mini NiFi
- Memcached
- Cassandra
- Difference between NiFi and Mini-NiFi
- Reference:

This post explains about how to build different BigData packages like Flink, Memcached, Kafka, Oozie, Kafka Streams, and Cassandra. You need to install Maven v3.3.9 and Open JDK8u.

About BigData components
There are many BigData components individually.

Build and Test Instructions

Install OpenJDK 8

```bash
$ wget http://openjdk.linaro.org/releases/jdk8u-server-release-1708.tar.xz
$ tar xvf jdk8u-server-release-1708.tar.xz
$ cd jdk8u-server-release-1708
$ export JAVA_HOME=$PWD
$ cd jre/lib/security/
$ rm cacerts

(for CentOS 7) $ ln --symbolic /etc/pki/java/cacerts .
(for Debian Jessie) $ ln --symbolic /etc/ssl/certs/java/cacerts .

$ cd jdk8u-server-release-1708/bin
$ export PATH=$PWD:$PATH
```

Install Maven v3.3.9
Download and export the Maven v3.3.9

```bash
$ wget http://www.us.apache.org/dist/maven/maven-3/3.3.9/binaries/apache-maven-3.3.9-bin.tar.gz
$ tar -xvf apache-maven-3.3.9-bin.tar.gz
$ cd apache-maven-3.3.9/bin
$ export PATH=$PWD:$PATH
```

Flink
The flink package build instructions are LEG-250

```bash
$ git clone https://github.com/apache/flink.git
$ cd flink
$ mvn clean package -DskipTests
```
Kafka

The kafka package build instructions are **LEG-252** You need to install "gradle" before executing any other instructions. The gradle install instructions are purely based on your distribution.

```bash
$ git clone https://github.com/apache/kafka.git
$ cd kafka
$ ./gradlew jar
```

Oozie

The oozie package build instructions are **LEG-276**

```bash
$ git clone https://github.com/apache/oozie.git
$ cd oozie
$ mvn clean package -DskipTests

$ cd minitest
$ mvn clean test
```

Nifi

The nifi package build instructions are **LEG-677**
$ git clone https://github.com/apache/nifi.git
$ cd nifi
$ mvn clean package -DskipTests

Testing:
-------

Mini NiFi

The mini-nifi package build instructions are in Jira issue

Apache MiNiFi has accepted the Bootstrap Config File: /home/nareshbhat/example-minifi-deploy/minifi-0.5.0-SNAPSHOT/conf/bootstrap.conf

Java home: /home/nareshbhat/Apache-packages-compilation/JDK/jdk8u-server-release-1708

nareshbhat@debian:~/example-minifi-deploy/minifi-0.5.0-SNAPSHOT/logs org.apache.nifi.minifi.MiNiFi

nareshbhat@debian:~$ ./bin/minifi.sh stop

nareshbhat@debian:~$ cd ~/example-minifi-deploy/minifi-0.5.0-SNAPSHOT

nareshbhat@debian:~$
Memcached

The memcached package build instructions are LEG-686

Install dependency package

$ sudo apt-get install libevent-dev
$ sudo apt-get update
$ sudo apt-get install mysql-server php5-mysql php5 php5-memcached

Clone and build memcached

$ git clone https://github.com/memcached/memcached.git
$ cd memcached
$ ./autogen.sh
$ ./configure
$ make all

You can run the testapp

$ ./testapp

You can check memcached version and run

$ ./memcached -V
memcached 1.5.4

I have started the memcached service on on console with below command

nareshbhat@debian:~/memcached$ memcached start

and test it on another console

nareshbhat@debian:~$ sudo netstat -tupan | grep 11211
tcp 0 0 0.0.0.0:11211 0.0.0.0:* LISTEN 20423/memcached
tcp6 0 0 :::11211 :::* LISTEN 20423/memcached
udp 0 0 0.0.0.0:11211 0.0.0.0:* 20423/memcached
udp6 0 0 :::11211 :::* 20423/memcached
nareshbhat@debian:~$

To verify memcached is recognized by the web server:

1. Create a phpinfo.php file in the web server’s docroot:

$ sudo nano /var/www/html/phpinfo.php

=========phpinfo.php start==========
<?php
// Show all information, defaults to INFO_ALL
phpinfo();
=========phpinfo.php end==========

2. Go to that page in your web browser.
For example, http://<IP-address>/phpinfo.php

3. Make sure memcached displays all the information on web browser

I have also executed database_test.php, cache_test.php

Cassandra

The cassandra package build instructions are LEG-695 You have to install ant tool to build cassandra.
$ git clone https://github.com/apache/cassandra.git
$ cd cassandra
$ git tag
$ git checkout cassandra-3.11.1
$ ant clean
$ ant

After successful build you can run the cassandra using command

$ ./bin/cassandra -f

On another terminal I run the below example

$ bin/cqlsh

Connected to Test Cluster at 127.0.0.1:9042.
[cqlsh 5.0.1 | Cassandra 3.11.1-SNAPSHOT | CQL spec 3.4.4 | Native protocol v4]
Use HELP for help.
cqlsh> CREATE SCHEMA schem1
... WITH replication = { 'class' : 'SimpleStrategy', 'replication_factor' : 1 };
cqlsh> USE schem1;
cqlsh:schem1> CREATE TABLE users {
... user_id varchar PRIMARY KEY,
... first varchar,
... last varchar,
... age int
... };
cqlsh:schem1> INSERT INTO users (user_id, first, last, age)
... VALUES ('jsmith', 'John', 'Smith', 42);
cqlsh:schem1> SELECT * FROM users;

user_id | age | first | last
--------+-----+-------+-------
jsmith | 42  | John  | Smith

A great example of the power and usefulness of MiNiFi is a recent demo at the TU-Automotive Detroit exhibition, where MiNiFi was loaded onto a custom Qualcomm modem located in a "connected car". As the car drives, massive amounts of data are generated by components throughout the car and routed via the CANBUS to be processed. Some data is important to stream back to a remote processing center in real-time -- this data is transmitted via an LTE connection. LTE is widely available, but bandwidth is expensive. Meanwhile, data that was much larger but less time-relevant (system diagnostics, etc.) could be batched and compressed, and then sent in bursts over WiFi when the car was in range of a known hotspot. MiNiFi coordinated all of the flow decisions and routing via geo-enrichment and control plane feedback. Here is a short video of Joe Niemiec explaining the process and showing the flow.

You can extrapolate that demo to many other use cases. It is helpful to think of MiNiFi as a "good guest" -- a lightweight agent that runs on hardware that is probably dedicated to a different primary purpose. Whether this is IoT, a cash register/point of sale system, a car modem, physical sensors, etc., is irrelevant to MiNiFi -- its job is to process and exfill this data while not taking unnecessary resources from the primary function. Contrast this with NiFi, which again, can run simultaneously with other applications, but ideally it has dedicated resources which it can maximize for its own performance.

Difference between NiFi and Mini-NiFi

Apache NiFi is a robust and secure framework for routing, transforming, and delivering data across a multitude of systems. NiFi can run in parallel with other applications, but it performs best when the entire system (or multiple systems in a cluster) are dedicated to it. It often uses SAN or RAID storage at the TB level for the massive amounts of content it ingests and the provenance it generates. The UI allows multiple users to quickly modify flows simultaneously on the same machine or across a cluster. The latest release candidate of NiFi (1.1.0 RC1) includes over 170 processors for custom integration with various systems and operations, and is 762 MB when compressed for download. In other words, NiFi is a server-class application.

Apache MINiFi was developed out of a recognized need to bring the capabilities of NiFi to the “edge” as “agents” -- accessing data from IoT and desktop-level devices, and applying primary features of NiFi at the earliest possible stage. Now data can be collected from a variety of protocols, have data provenance generated immediately for more holistic governance and transparency, have light transformations applied at source, be encrypted, be prioritized, and be redundantly routed back to the more powerful transformations done in the cloud or data center.

Now, all of these behaviors can be performed with custom scripts, but then the problem of command & control (C2) is encountered. With hundreds, thousands, or even millions of these devices existing, how can each be monitored and exfilled, and what happens when the flow needs to change? It could be to report back to a new endpoint, to update the frequency at which it is collected or transmitted, or to handle new metrics or metadata from the device. This manual process does not scale. With MiNiFi’s integration with NiFi, a flow can be developed using the UI in NiFi and transparently translated to a MiNiFi flow and pushed out to classes of agents across the world.

With manual modification to remove unnecessary processors and features, NiFi can be trimmed to fit on a Raspberry Pi. But it still requires the JVM, and there are plenty of devices that won’t support it. MiNiFi is offered in Java and C++, and the footprint is on a completely different scale -- 39 MB for the Java agent (tar) and 310K for the C++ agent (tar).

A great example of the power and usefulness of MiNiFi is a recent demo at the TU-Automotive Detroit exhibition, where MiNiFi was loaded onto a custom Qualcomm modem located in a "connected car". As the car drives, massive amounts of data are generated by components throughout the car and routed via the CANBUS to be processed. Some data is important to stream back to a remote processing center in real-time -- this data is transmitted via an LTE connection. LTE is widely available, but bandwidth is expensive. Meanwhile, data that was much larger but less time-relevant (system diagnostics, etc.) could be batched and compressed, and then sent in bursts over WiFi when the car was in range of a known hotspot. MiNiFi coordinated all of the flow decisions and routing via geo-enrichment and control plane feedback. Here is a short video of Joe Niemiec explaining the process and showing the flow.

You can extrapolate that demo to many other use cases. It is helpful to think of MiNiFi as a "good guest" -- a lightweight agent that runs on hardware that is probably dedicated to a different primary purpose. Whether this is IoT, a cash register/point of sale system, a car modem, physical sensors, etc., is irrelevant to MiNiFi -- its job is to process and exfill this data while not taking unnecessary resources from the primary function. Contrast this with NiFi, which again, can run simultaneously with other applications, but ideally it has dedicated resources which it can maximize for its own performance.
Reference:
http://devdocs.magento.com/guides/v2.0/config-guide/memcache/memcache_ubuntu.html
https://github.com/linux-on-ibm-z/docs/wiki/Building-Apache-Cassandra-3.11
https://github.com/apache/cassandra