Consumer Group Example

Using the High Level Consumer

Why use the High Level Consumer

Sometimes the logic to read messages from Kafka doesn't care about handling the message offsets, it just wants the data. So the High Level Consumer is provided to abstract most of the details of consuming events from Kafka.

First thing to know is that the High Level Consumer stores the last offset read from a specific partition in ZooKeeper. This offset is stored based on the name provided to Kafka when the process starts. This name is referred to as the Consumer Group.

The Consumer Group name is global across a Kafka cluster, so you should be careful that any 'old' logic Consumers be shutdown before starting new code. When a new process is started with the same Consumer Group name, Kafka will add that processes' threads to the set of threads available to consume the Topic and trigger a 're-balance'. During this re-balance Kafka will assign available partitions to available threads, possibly moving a partition to another process. If you have a mixture of old and new business logic, it is possible that some messages go to the old logic.

Designing a High Level Consumer

The first thing to know about using a High Level Consumer is that it can (and should!) be a multi-threaded application. The threading model revolves around the number of partitions in your topic and there are some very specific rules:

- if you provide more threads than there are partitions on the topic, some threads will never see a message
- if you have more partitions than you have threads, some threads will receive data from multiple partitions
- if you have multiple partitions per thread there is NO guarantee about the order you receive messages, other than that within the partition the offsets will be sequential. For example, you may receive 5 messages from partition 10 and 6 from partition 11, then 5 more from partition 10 followed by 5 more from partition 10 even if partition 11 has data available.
- adding more processes/threads will cause Kafka to re-balance, possibly changing the assignment of a Partition to a Thread.

Next, your logic should expect to get an iterator from Kafka that may block if there are no new messages available.

Here is an example of a very simple consumer that expects to be threaded.

```
package com.test.groups;

import kafka.consumer.ConsumerIterator;
import kafka.consumer.KafkaStream;

public class ConsumerTest implements Runnable {
    private KafkaStream m_stream;
    private int m_threadNumber;

    public ConsumerTest(KafkaStream a_stream, int a_threadNumber) {
        m_threadNumber = a_threadNumber;
        m_stream = a_stream;
    }

    public void run() {
        ConsumerIterator<byte[], byte[]> it = m_stream.iterator();
        while (it.hasNext())
            System.out.println("Thread " + m_threadNumber + ": " + new
String(it.next().message()));
        System.out.println("Shutting down Thread: " + m_threadNumber);
    }
}
```

The interesting part here is the **while (it.hasNext())** section. Basically this code reads from Kafka until you stop it.

Configuring the test application

Unlike the SimpleConsumer the High level consumer takes care of a lot of the bookkeeping and error handling for you. However you do need to tell Kafka where to store some information. The following method defines the basics for creating a High Level Consumer:

```
private static ConsumerConfig createConsumerConfig(String a_zookeeper, String
a_groupId) {
        Properties props = new Properties();
        props.put("zookeeper.connect", a_zookeeper);
        props.put("group.id", a_groupId);
        props.put("zookeeper.session.timeout.ms", "400");
        props.put("zookeeper.sync.time.ms", "200");
        props.put("auto.commit.interval.ms", "1000");
        return new ConsumerConfig(props);
    }
```

The '**zookeeper.connect**' string identifies where to find once instance of Zookeeper in your cluster. Kafka uses ZooKeeper to store offsets of messages consumed for a specific topic and partition by this Consumer Group

The '**group.id**' string defines the Consumer Group this process is consuming on behalf of.

The '**zookeeper.session.timeout.ms**' is how many milliseconds Kafka will wait for ZooKeeper to respond to a request (read or write) before giving up and continuing to consume messages.

The '**zookeeper.sync.time.ms**' is the number of milliseconds a ZooKeeper 'follower' can be behind the master before an error occurs.

The '**auto.commit.interval.ms**' setting is how often updates to the consumed offsets are written to ZooKeeper. Note that since the commit frequency is time based instead of # of messages consumed, if an error occurs between updates to ZooKeeper on restart you will get replayed messages.

More information about these settings can be found here

Creating the thread pool

This example uses the Java *java.util.concurrent* package for thread management since it makes creating a thread pool very simple.

```
    public void run(int a_numThreads) {
        Map<String, Integer> topicCountMap = new HashMap<String, Integer>();
        topicCountMap.put(topic, new Integer(a_numThreads));
        Map<String, List<KafkaStream<byte[], byte[]>>> consumerMap =
consumer.createMessageStreams(topicCountMap);
        List<KafkaStream<byte[], byte[]>> streams = consumerMap.get(topic);


        // now launch all the threads
        //
        executor = Executors.newFixedThreadPool(a_numThreads);

        // now create an object to consume the messages
        //
        int threadNumber = 0;
        for (final KafkaStream stream : streams) {
            executor.execute(new ConsumerTest(stream, threadNumber));
            threadNumber++;
        }
    }
```

First we create a Map that tells Kafka how many threads we are providing for which topics. The consumer.createMessageStreams is how we pass

this information to Kafka. The return is a map of KafkaStream to listen on for each topic. (Note here we only asked Kafka for a single Topic but we could have asked for multiple by adding another element to the Map.)

Finally we create the thread pool and pass a new ConsumerTest object to each thread as our business logic.

Clean Shutdown and Error Handling

Kafka does not update Zookeeper with the message offset last read after every read, instead it waits a short period of time. Due to this delay it is possible that your logic has consumed a message and that fact hasn't been synced to zookeeper. So if your client exits/crashes you may find messages being replayed next time to start.

Also note that sometimes the loss of a Broker or other event that causes the Leader for a Partition to change can also cause duplicate messages to be replayed.

To help avoid this, make sure you provide a clean way for your client to exit instead of assuming it can be 'kill -9'd.

As an example, the main here sleeps for 10 seconds, which allows the background consumer threads to consume data from their streams 10 seconds. Since auto commit is on, they will commit offsets every second. Then, shutdown is called, which calls shutdown on the consumer, then on the ExecutorService, and finally tries to wait for the ExecutorService to finish all outsanding work. This gives the consumer threads time to finish processing the few outstanding messages that may remain in their streams. Shutting down the consumer causes the iterators for each stream to return false for hasNext() once all messages already received from the server are processed, so the other threads should exit gracefully. Additionally, with auto commit enabled, the call to consumer.shutdown() will commit the final offsets.

```
try {
            Thread.sleep(10000);
        } catch (InterruptedException ie) {

        }
        example.shutdown();
```

In practice, a more common pattern is to use sleep indefinitely and use a shutdown hook to trigger clean shutdown.

Running the example

The example code expects the following command line parameters:

- ZooKeeper connection string with port number
- Consumer Group name to use for this process
- Topic to consume messages from
- # of threads to launch to consume the messages

For example:

```
server01.myco.com1:2181 group3 myTopic  4
```

Will connect to port 2181 on *server01.myco.com* for ZooKeeper and requests all partitions from Topic *myTopic* and consume them via *4* threads. The Consumer Group for this example is *group3*.

Full Source Code

```
package com.test.groups;

import kafka.consumer.ConsumerConfig;
import kafka.consumer.KafkaStream;
import kafka.javaapi.consumer.ConsumerConnector;

import java.util.HashMap;
```

```
import java.util.List;
import java.util.Map;
import java.util.Properties;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ConsumerGroupExample {
    private final ConsumerConnector consumer;
    private final String topic;
    private  ExecutorService executor;

    public ConsumerGroupExample(String a_zookeeper, String a_groupId, String a_topic)
{
        consumer = kafka.consumer.Consumer.createJavaConsumerConnector(
                createConsumerConfig(a_zookeeper, a_groupId));
        this.topic = a_topic;
    }

    public void shutdown() {
        if (consumer != null) consumer.shutdown();
        if (executor != null) executor.shutdown();
        try {
            if (!executor.awaitTermination(5000, TimeUnit.MILLISECONDS)) {
                System.out.println("Timed out waiting for consumer threads to shut
down, exiting uncleanly");
            }
        } catch (InterruptedException e) {
            System.out.println("Interrupted during shutdown, exiting uncleanly");
        }
   }

    public void run(int a_numThreads) {
        Map<String, Integer> topicCountMap = new HashMap<String, Integer>();
        topicCountMap.put(topic, new Integer(a_numThreads));
        Map<String, List<KafkaStream<byte[], byte[]>>> consumerMap =
consumer.createMessageStreams(topicCountMap);
        List<KafkaStream<byte[], byte[]>> streams = consumerMap.get(topic);

        // now launch all the threads
        //
        executor = Executors.newFixedThreadPool(a_numThreads);

        // now create an object to consume the messages
        //
        int threadNumber = 0;
        for (final KafkaStream stream : streams) {
            executor.submit(new ConsumerTest(stream, threadNumber));
            threadNumber++;
        }
    }

    private static ConsumerConfig createConsumerConfig(String a_zookeeper, String
a_groupId) {
        Properties props = new Properties();
        props.put("zookeeper.connect", a_zookeeper);
        props.put("group.id", a_groupId);
        props.put("zookeeper.session.timeout.ms", "400");
        props.put("zookeeper.sync.time.ms", "200");
        props.put("auto.commit.interval.ms", "1000");
```

```
        return new ConsumerConfig(props);
    }

    public static void main(String[] args) {
        String zooKeeper = args[0];
        String groupId = args[1];
        String topic = args[2];
        int threads = Integer.parseInt(args[3]);

        ConsumerGroupExample example = new ConsumerGroupExample(zooKeeper, groupId,
topic);
        example.run(threads);

        try {
            Thread.sleep(10000);
        } catch (InterruptedException ie) {

        }
        example.shutdown();
```

```
    }
}
```