Old dog, New tricks
A data engineering blog

Moving binary data with Kafka

Posted on September 20, 2016 by Guy Shilo

Last time we saw how to send text messages with Kafka. This time we will use Kafka to move binary data around.

Our test case will be monitoring an input directory for new files, and whenever a new file is detected we will read it and publish it to a Kafka topic.

The consumer on the other end will receive the files and write them to a target directory.

This all works fine when we have small files, but I want it to be able to transfer large files as well. Kafka is not designed to move large messages and it is not the best choice for moving large binary files like photos. In her article Handling large messages in Kafka Gwen Shapira suggests several ways to deal with large messages and says that the optimal message size for Kafka is around 10k. So I decided to go for the splitting solution and split each file I send to 10k chunks before sending it to Kafka. [Added on February 21 2018] Lately I investigated messages size deeper and published "Finding Kafka optimal message size".

The consumer has to assemble the file from its chunks. The simple approach I chose here is to create a topic with only one partition. This way I can be sure that the chunks arrive to the consumer in the same order they were produced. This does not scale and may not suffice in a very busy production system. Using multiple partition with parallel publishing and consuming will be more robust but will require much more complex logic on the consumer side to be able to assemble the file from non-ordered chunks. Just for the demonstration I will stick with the simpler solution. After a long time, on December 2017, I published an improved version that tries to address those problems.

first, lets create the topic:

```
kafka-topics.sh --zookeeper hadoop:2181 --create --topic filestopic --partitions 1 -
Created topic "filestopic".
```

Both producer and consumer read their parameters from a configuration file. Some parameters are common and others are unique for producer or consumer side. Here is a unified configuration file that can be used with both of them:

```
# this should point to some Kafka brokers
bootstrap.servers=192.168.56.101:9092
topic=filestopic
# We are moving files from Windows to Linux
watchdir=c:\watchdir
outdir=/home/kafka/outdir
group.id=test
```

Here is the Config class which is common to both producer and consumer:

```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.HashMap;

/**
*
* @author guys
*/
public class Config {

// this class reads the configuration parameters from the config file and serves the
HashMap&lt;String, String&gt; conf;

public Config(String filePath) throws FileNotFoundException, IOException
{
```

```
conf=new HashMap();
File file=new File(filePath);
BufferedReader br=new BufferedReader(new FileReader(file));
String[] vals;
String line=br.readLine();
while (line!=null)
{
if (!line.startsWith("#"))
{
vals=line.toLowerCase().split("=");
conf.put(vals[0], vals[1]);
}
line=br.readLine();
}

}

public String get(String key)
{
return conf.get(key.toLowerCase());
}

}
```

This is the producer code which uses Java Watch API to monitor a directory for new files:

```
package producers;

import java.io.ByteArrayOutputStream;
import java.io.File;
import java.io.IOException;
import java.nio.file.FileSystems;
import java.nio.file.Files;
import java.nio.file.Path;
import static java.nio.file.StandardWatchEventKinds.ENTRY_CREATE;
import java.nio.file.WatchEvent;
import java.nio.file.WatchKey;
import java.nio.file.WatchService;
import java.util.ArrayList;
```

```
import java.util.Arrays;
import java.util.Properties;
import java.util.logging.Level;
import java.util.logging.Logger;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.ProducerRecord;
/**
*
* @author guys
*/
public class BinaryProducer {
Config myConf;
Producer&lt;String, byte[]&gt; producer;
String topic, bootstrapServers, watchDir;
Path path;
ByteArrayOutputStream out;


public BinaryProducer(String configPath) throws IOException
{
// Read initial configuration
myConf=new Config(configPath);

// setting the kafka producer stuff
Properties props = new Properties();
props.put("bootstrap.servers",myConf.get("bootstrap.servers"));
props.put("acks", "1");
props.put("compression.type","snappy");
props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer"
props.put("value.serializer", "org.apache.kafka.common.serialization.ByteArraySerial
producer = new KafkaProducer&lt;&gt;(props);

topic=myConf.get("topic");
watchDir=myConf.get("watchdir");
path=FileSystems.getDefault().getPath(watchDir);


}

// Takes a whole binary file content and splits it into 10k chunks
```

```
private ArrayList splitFile(String name,byte[] datum)
{
int i,l=datum.length;
int block=10240;
int numblocks=l/block;
int counter=0, totalSize=0;
int marker=0;
byte[] chunk;
ArrayList&lt;byte[]&gt; data=new ArrayList();
for (i=0;i&lt;numblocks;i++)
{
counter++;
chunk=Arrays.copyOfRange(datum, marker, marker+block);
data.add(chunk);
totalSize+=chunk.length;
marker+=block;
}
chunk=Arrays.copyOfRange(datum,marker,l);
data.add(chunk);
// the null value is a flag to the consumer, specifiying that it has reached the end
data.add(null);
return data;
}

private void start() throws IOException, InterruptedException
{
String fileName;
byte[] fileData;
ArrayList&lt;byte[]&gt; allChunks;
// the watcher watches for filesystem changes
WatchService watcher = FileSystems.getDefault().newWatchService();
WatchKey key;
path.register(watcher, ENTRY_CREATE);

while (true)
{
key = watcher.take();
// The code gets beyond this point only when a filesystem event occurs

for (WatchEvent&lt;?&gt; event: key.pollEvents())
{
```

```
WatchEvent.Kind&lt;?&gt; kind = event.kind();
// We act only if a new file was added
if (kind==ENTRY_CREATE)
{
WatchEvent&lt;Path&gt; ev = (WatchEvent&lt;Path&gt;)event;
Path filename = ev.context();
fileName=filename.toString();
// We need this little delay to make sure the file is closed before we read it
Thread.sleep(500);

fileData=Files.readAllBytes(FileSystems.getDefault().getPath(watchDir+File.separator
allChunks=splitFile(fileName,fileData);
for (int i=0;i&lt;allChunks.size();i++)
{
publishMessage(fileName,(allChunks.get(i)));
}
System.out.println("Published file "+fileName);

}
}
key.reset();
}
}

private void publishMessage(String key, byte[] bytes)
{
ProducerRecord &lt;String, byte[]&gt; data =new ProducerRecord&lt;&gt;(topic, key, b
producer.send(data);

}

public static void main (String args[])
{
BinaryProducer abp;
try {
abp=new BinaryProducer(args[0]);
try {
abp.start();
} catch (InterruptedException ex) {
Logger.getLogger(BinaryProducer.class.getName()).log(Level.SEVERE, null, ex);
}
```

```
} catch (IOException ex) {
Logger.getLogger(BinaryProducer.class.getName()).log(Level.SEVERE, null, ex);
}
}
}
```

This is the consumer:

```
package consumers;

import java.io.ByteArrayOutputStream;
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.Arrays;
import java.util.Properties;
import java.util.logging.Level;
import java.util.logging.Logger;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;

/**
*
* @author guys
*/
public class BinaryConsumer {
Properties props;
KafkaConsumer&lt;String, byte[]&gt; consumer;
Config myConf;
String outDir;

/**
*
*/

public BinaryConsumer(String confPath) throws IOException
```

```
{
// setting Kafka configuration
myConf=new Config(confPath);
props = new Properties();
props.put("bootstrap.servers", myConf.get("bootstrap.servers"));
props.put("group.id", myConf.get("group.id"));
props.put("enable.auto.commit", "true");
props.put("compression.type","snappy");
props.put("fetch.message.max.bytes","7340032");
props.put("auto.commit.interval.ms", "1000");
props.put("session.timeout.ms", "30000");
props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeseriali
props.put("value.deserializer", "org.apache.kafka.common.serialization.ByteArrayDese
outDir=myConf.get("outdir");

}

public void createConsumer()
{
consumer = new KafkaConsumer&lt;&gt;(props);
consumer.subscribe(Arrays.asList(myConf.get("topic")));
}

public void start() throws IOException
{
String name=null;
ByteArrayOutputStream bos=new ByteArrayOutputStream();
//Loop endlessly and poll for new kafka records.
while (true) {
ConsumerRecords&lt;String, byte[]&gt; records = consumer.poll(100);
for (ConsumerRecord&lt;String, byte[]&gt; record : records)
{
// Test weather we have reached EOF
if (record.value()==null)
{
// we reached EOF, write it to disk.
System.out.println("Writing file "+name);
writeFile(name,bos.toByteArray());
bos.reset();
}
else
```

```
{
// this is just another chunk, accumulate it in memory
name=record.key();
bos.write(record.value());
}
}
}
}


private void writeFile(String name,byte[] rawdata) throws IOException
{

File f=new File(outDir);
if (!f.exists())
f.mkdirs();

FileOutputStream fos=new FileOutputStream(outDir+File.separator+name);
fos.write(rawdata);
fos.flush();
fos.close();


}


public static void main(String[] args) {
try {
BinaryConsumer abc;
abc=new BinaryConsumer(args[0]);
abc.createConsumer();
abc.start();
} catch (IOException ex) {
Logger.getLogger(BinaryConsumer.class.getName()).log(Level.SEVERE, null, ex);
}

}

}
```

You will need the following jar libraries in order to compile and run it:

commons-compress-1.8.1.jar

paranamer-2.7.jar

slf4j-api-1.7.7.jar

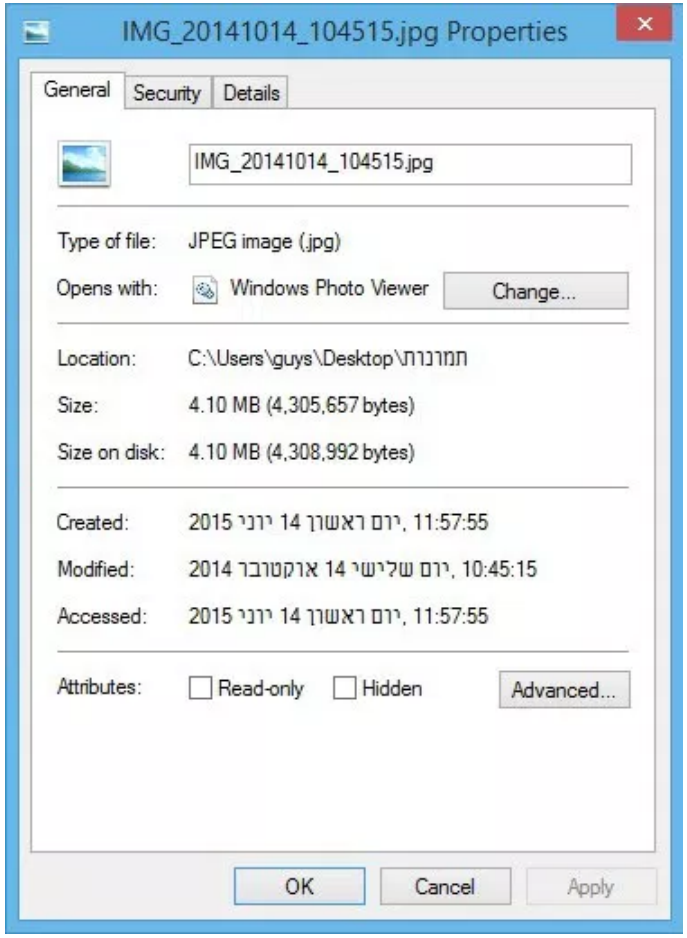
snappy-java-1.1.1.3.jar

xz-1.5.jar

kafka-clients-0.10.0.0.jar (If you are using Kafka that comes with CDH 5 then you will need version 0.9.0 of this jar).

It is hard to demonstrate this in action using only words and screenshots but I will do my best. The producer will run on my Windows laptop while the consumer will run on the Linux server where Kafka broker runs.

This is the file that I want to transfer as it is on the source Windows system:

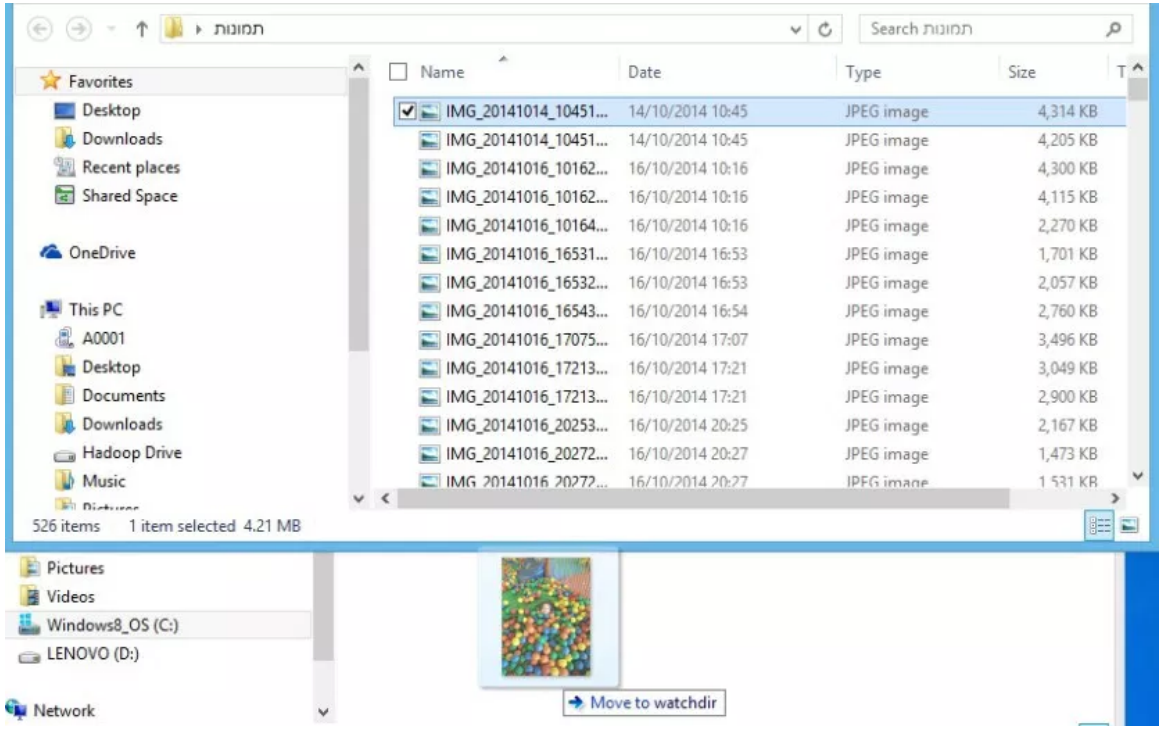


The file name is IMG_20141014_104515.jpg and its size is 4.1MB (4305657 bytes).

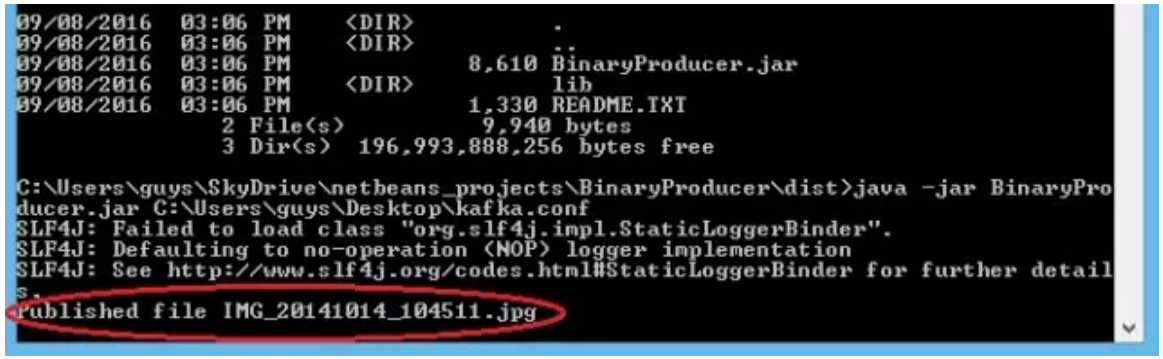
After starting the producer with

```
java -jar BinaryProducer.jar &lt;path to config file&gt;
```

we drag the file we mentioned above to the monitored directory on Windows:



The producer verifies it has picked up the new file and sent it to Kafka:



Now, on the consumer side, we run the consumer:

```
java -jar BinartConsumer.jar /home/kafka/kafka.conf
```

As you can see below, the consumer has written the file to disk. The file name is IMG_20141014_104515.jpg and its size is 4305657 bytes, just as the source (I was also able to open and see the photo).

```
[kafka@hadoop ~]$ java -jar BinaryConsumer.jar /home/kafka/kafka.conf
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
Writing file IMG_20141014_104515.jpg
^C[kafka@hadoop ~]$
[kafka@hadoop ~]$ cd outdir
[kafka@hadoop outdir]$ ls -l
total 4208
-rw-r--r-- 1 kafka hadoop 4305657 Sep 15 16:30 IMG_20141014_104515.jpg
[kafka@hadoop outdir]$
```

The transfer succeeded and the file was reassembled successfully on the consumer side.

Related

Moving binary data with Kafka - a more realistic scenario

Finding Kafka optimal message size
February 18, 2018

Monitoring Kafka consumer lag with Burrow
January 16, 2017

December 24, 2017 In "Kafka" In "Kafka" In "Kafka"

This entry was posted in Kafka and tagged binary, java, kafka. Bookmark the permalink.

5 Responses to *Moving binary data with Kafka*

Guy Shilo *says:*

December 26, 2017 at 9:20 pm

It's been a long time since I published this and I have done many different things since then. But lately I was working on a better, more scalable version, and today I published it in a new post: Moving binary data with Kafka – a more realistic scenario

Reply

Jackie Weng *says:*

December 1, 2017 at 12:35 am

Since you are using the filename as the key, even with multiple partitions, all messages will actually end up go to the same partition, meaning they will be ordered. Assuming the topic is only used for one single folder, there should be no filename conflicts.

This behaviour means using multiple partitions does not help with single large file, but would provide scaling when multiple files are added.

Reply

archenroot *says:*

November 16, 2017 at 8:59 pm

I think you can enhance the binary chunks with kind of binary header saying (I am part 3 of stream 234) and that way you can publish the chunks in parallel accross multiple partitions, so you can scale just fine. The consumer is responsible to create buffers for different binarry streams, once the stream is completed meaning all chunks in buffer (can be some persistence buffer with lets say some key,value store to track overview about the buffer). So you don't care about the order.

Reply

Guy Shilo *says:*

November 16, 2017 at 10:46 pm

You are absolutely right. This version is not good for real production load because it's not parallel. I wanted to write a better version and present it in another post but did not find the time for it yet…

Reply

archenroot *says:*

November 21, 2017 at 12:19 pm

This concept is generic actually to be applied to any kind of broker. ActiveMQ Artemis have this built in, it is capable to transport gigabyte files with 50mb of ram usage…

Reply

This site uses Akismet to reduce spam. Learn how your comment data is processed.

Old dog, New tricks

Proudly powered by WordPress.