Assignment 4: Kafka Streams

Due date: Tuesday December 7th at 11:00pm Waterloo time

ECE 454/751: Distributed Computing

Instructor: Dr. Wojciech Golab
wgolab@uwaterloo.ca

A few house rules

Collaboration:

- groups of 1, 2 or 3

Managing source code:

- do keep a backup copy of your code outside of ecelinux
- do not post your code in a public repository (e.g., GitHub free tier)

Software environment:

- test on eceubuntu and ecetesla
- code using Java 11
- Kafka 2.4.1, client code provided on ecelinux and server hosted on manta.uwaterloo.ca
- Apache ZooKeeper 3.4.13, hosted on manta.uwaterloo.ca

Overview

- In this assignment, you will implement a simple real-time stream processing application using the **Kafka Streams API**.
- A partial implementation of the application is provided in the starter code tarball.
- Your goal is to complete the code so that it meets the functional and non-functional requirements.
- ZooKeeper and Kafka are provided on manta.uwaterloo.ca on the default ports (2181 and 9092, respectively).
- ZooKeeper is provided only because it is needed by Kafka. Your Java code should not interact with ZooKeeper directly. Use Kafka utilities to create topics, reset applications, etc.

Learning objectives

Upon successful completion of the assignment, you will know how to:

- process incoming messages from a topic and output messages to a topic using the **Kafka Streams API** (not to be confused with the Streams Processor API).
- use functional programming to manipulate streams
- use Kafka state stores
- use Kafka serializers and deserializers

Inputs and outputs

Your Kafka application will consume input records from two topics, and will produce output records to a third topic. The names of the topics will be provided at runtime as command line arguments to your program. Each Kafka topic will have a single partition.

Note: The variable $USER denotes your Nexus user ID.

Inputs

- Your application will consume records from two Kafka topics containing information regarding students and classrooms, respectively.
- The first topic provides info on students and their whereabouts. Each message is a string key-value pair of the following form:

  Student_ID,Room_ID
- The second topic provides info on classrooms and their capacity. Each message is a string key-value pair of the following form:

  Room_ID,max_capacity
- The key and value are separated by one comma, with no whitespace around the comma.

Input examples: classroom topic

- In the following example, RoomA is assigned a maximum capacity of 10, then RoomB is assigned a maximum capacity of 10, and finally the maximum capacity of RoomB is increased from 10 to 15.
- Sequence of messages written to classroom topic:

  RoomA,10
  RoomB,10
  RoomB,15

- **Note 1:** The capacity of a room can increase, as shown above, and it can also decrease.
- **Note 2:** If not specified, the capacity of a room is unlimited.

Input examples: student topic

- In the following example, Student100 enters RoomB, then Student200 enters RoomB, and finally Student100 moves from RoomB to RoomA.
- Sequence of key-value messages written to student topic:

  Student100,RoomB

  Student200,RoomB

  Student100,RoomA

- **Note 1:** A student can be in at most one room at any given time. The student topic only shows which room is entered.
- **Note 2:** A student can enter a room that is already at capacity or above capacity. Thus, rooms can overfill.

Output

- The goal of the application is to output (i.e., commit to the output topic) the names of rooms for which the current occupancy exceeds the maximum capacity, along with some additional information that depends on the particular state change.
- Each time a state change in occupancy or capacity occurs such that the occupancy of a room exceeds its capacity after the state change, the application should output a string key-value pair of the following form:

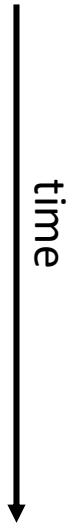
  Room_ID,occupancy_after_state_change

- Each time a state change in occupancy or capacity occurs such that the occupancy of a room was above its capacity before the state change, and is equal to or below its capacity after the state change, the application should output a string key-value pair of the following form:

  Room_ID,OK

Example of inputs and output

student topic	**classroom topic**	**output topic**
	RoomA,2	
	RoomB,2	
Student100,RoomA		
Student200,RoomA		
Student300,RoomA		
		RoomA,3
Student400,RoomA		
		RoomA,4
Student100,RoomB		
		RoomA,3
Student200,RoomB		
		RoomA,OK
Student300,RoomB		
		RoomB,3



Requirements

1. The application should be configured so that each input message is consumed and processed in two seconds or less. For example, if the occupancy of a room increases beyond the maximum capacity then the corresponding output record should be committed to the output topic within two seconds.
2. The implementation must tolerate crash failures. One of the cases tested by the grading script will crash your application (via kill -9) after publishing several messages to the input topics, then restart your application and publish more messages to the input topics. Your stream application must resume processing and publish messages to the output topic as if the crash failure did not occur.

Notes

1. The correct output depends not only on the messages supplied in the input streams, but also on the order in which student and classroom messages are processed. **The grading script will only publish to one topic at a time, in a single thread. It will wait at least two seconds between switching topics. It may switch topics back and forth several times.**
2. If a student enters a room for which there is no room data, then assume that the room's max capacity is unlimited.

Packaging and submission

- All your Java classes must be in the default package.
- You must use the Streams API. The mainline of your application goes in A4Application.java. A partial implementation and build script are provided. You may create additional Java files if you like.
- Do not change the structure of the command line arguments of the provided A4Application program.
- The classpath for this assignment comprises all the jar files packaged with Kafka under kafka_2.11-2.4.1/libs.
- Use the provided package.sh script to create a tarball for electronic submission, and upload it to the appropriate LEARN dropbox before the deadline.

Grading scheme

Evaluation structure:

Correctness of outputs: 100%

Penalties of up to 100% will apply in the following cases:

- the solution does not use the Kafka Streams API
- grading script cannot see the output because the application takes too long (i.e., five seconds instead of one or two seconds) to produce the output after a state change
- solution cannot be compiled or throws an exception during testing despite receiving valid input
- solution produces incorrect outputs
- solution is improperly packaged

Additional info and hints

Creating and resetting topics

- The two input topics and the output topic must be created prior to running your application.
- You may also want to purge the messages in these topics occasionally during testing.
- The starter code includes a script called reset_topics.sh that performs both functions. It first deletes all three topics and then (re)creates them.
- You may assume that each topic will have only one partition when your code is executed by the grading script.

Resetting your application

- Some stream processing applications use stateful operators, like *count*. These operators maintain state in a fault-tolerant manner using Kafka state stores.
- In addition, Kafka keeps track of which messages have been consumed by your application.
- You may want to reset this internal state prior to each run, which entails performing a local reset via the KafkaStreams class, as well as a global reset via the kafka-streams-application-reset.sh utility provided in Kafka.
- The starter code includes a script called reset_app.sh that performs both types of reset on your application.

Producing inputs

- The producer_student.sh and producer_classroom.sh scripts provided with the starter code are wrappers around the Kafka command line producer utility.
- They allow you to enter key-value pairs into the input topics using the console.
- The key and value are entered on one line, separated by a comma with no spaces around it.
- Remember to shut the producers down before running reset_topics.sh or reset_app.sh, as otherwise the input topics may not be reset correctly.

Consuming outputs

- The consumer.sh script provided with the starter code is a wrapper around the Kafka command line consumer utility.
- It allows you to dump the output of your application to the console.
- Remember to shut the consumer down before running reset_topics.sh or reset_app.sh, as otherwise the output topic may not be reset correctly.

Additional guidelines

- Use built-in Kafka Streams API features as much as possible instead of rolling your own code for fundamental stream operations.
- You may use the default state store for stateful stream operators. Do not bypass Kafka's state store by storing data (e.g., number of students in each classroom) in ordinary program variables.
- Setting the CACHE_MAX_BYTES_BUFFERING_CONFIG property to zero should ensure that the application produces outputs in a timely manner (i.e., around one second). This is already done for you in the mainline of the starter code.
- Using Kafka correctly will ensure that your application is fault tolerant. We will simulate client failures only during grading, and this will be done in Linux using kill -9 (SIGKILL).