Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Spring Term 2019 Dr. Vasiliki Kalavri

Data Stream Processing and Analytics

Assignment 9

In this assignment you will learn:
- How to define and use managed state in Flink applications
- How to enable checkpoints and recover from consistent state in case of failures
- How to trigger a savepoint and use it to safely re-configure an application

As we saw in the previous lecture, Flink's consistent checkpointing relies on *resettable* sources; sources must be capable of replaying data from a specified point in time or a given offset. One such source is Apache Kafka you are hopefully quite familiar with it by now.

For this assignment, you are going to use the code you wrote in Assignment #4. Create a new project and launch the following two applications:

- A **source program** that starts a `WikipediaEditsSource` and sends edit events to a Kafka topic. You should have implemented this already as Task 2 in Assignment #4.
- An **analysis program** that reads from the Kafka topic above and continuously aggregates the byte changes made per user. You can use a `RichFlatMapFunction` as shown below to keep track of the current byte diff counts per user in a `HashMap`:

```
// Read the edits input stream
DataStream<...> edits = env
      .addSource(kafkaConsumer)
      .setParallelism(1)
      // Your Kafka parser
      .flatMap(new ParseKafkaMessage());

// Partition by user and compute rolling diffs
edits.keyBy(0)
.flatMap(new ComputeDiffs())
```

```
.addSink(...);

==========================================================

// Keep track of user byte diffs in a HashMap
public static final class ComputeDiffs extends RichFlatMapFunction<
      Tuple2<String, Integer>, Tuple2<String, Integer>> {

  // user -> diffs
  private HashMap<String, Integer> diffs;

  @Override
  public void open(Configuration parameters) throws Exception {
     diffs = new HashMap<>();
  }

  @Override
  public void flatMap(Tuple2<String, Integer> in,
            Collector<Tuple2<String, Integer>> out) throws Exception {

     String user = in.f0;
     int diff = in.f1;

     if (diffs.containsKey(user)) {
        // Update existing HashMap value
        diffs.put(user, diffs.get(user) + diff);
     }
     else {
        // Insert new user in HashMap
        diffs.put(user, diff);
     }
     out.collect(new Tuple2<String, Integer>(user, diffs.get(user)));
  }
}
```

1. Cause a failure and set a restart strategy

Start the source program that sends wikipedia edit events to your Kafka topic and build a jar with the analysis program using Maven:

```
> mvn clean package
```

Submit it to a local Flink cluster (you will find the jar in the /target folder):

```
> ./bin/start-cluster.sh
> ./bin/flink run /path/to/your/jar
```

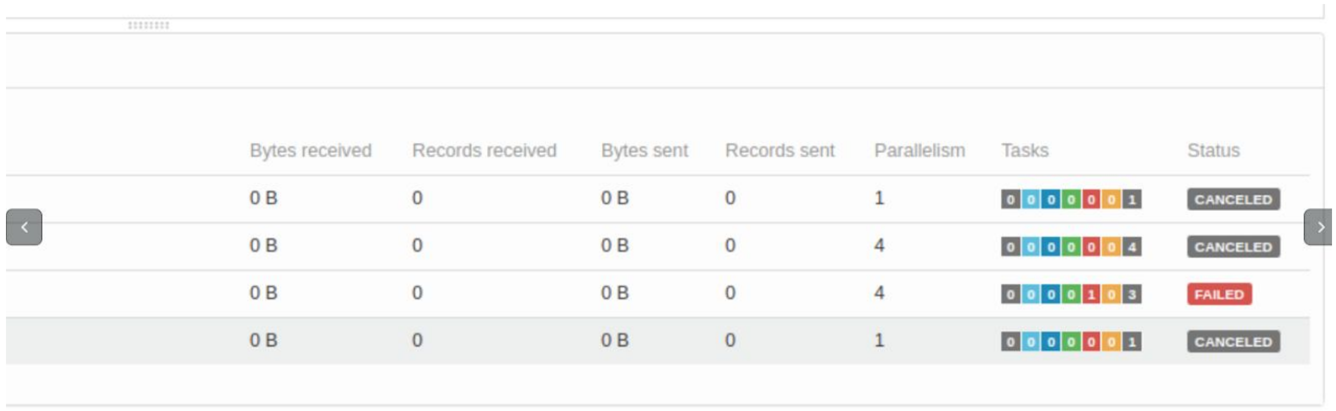
Visit Flink's Web UI http://localhost:8081 to check that your job is running correctly. Click on one of the running tasks. What is the *Attempt* number?

If you have used the default configuration, your local Flink cluster must be running with a single TaskManager. Let's find its process id and kill it:

```
> ps -ef | grep taskmanager
```

```
> kill -9 <pid>
```

After a few seconds, Flink will not be able to reach the TaskManager process and you will see your job failing:



Bytes received	Records received	Bytes sent	Records sent	Parallelism	Tasks	Status
0 B	0	0 B	0	1	0 0 0 0 0 0 1	CANCELED
0 B	0	0 B	0	4	0 0 0 0 0 0 4	CANCELED
0 B	0	0 B	0	4	0 0 0 0 1 0 3	FAILED
0 B	0	0 B	0	1	0 0 0 0 0 0 1	CANCELED

Now restart the TaskManager by running

```
> ./bin/taskmanager.sh start
```

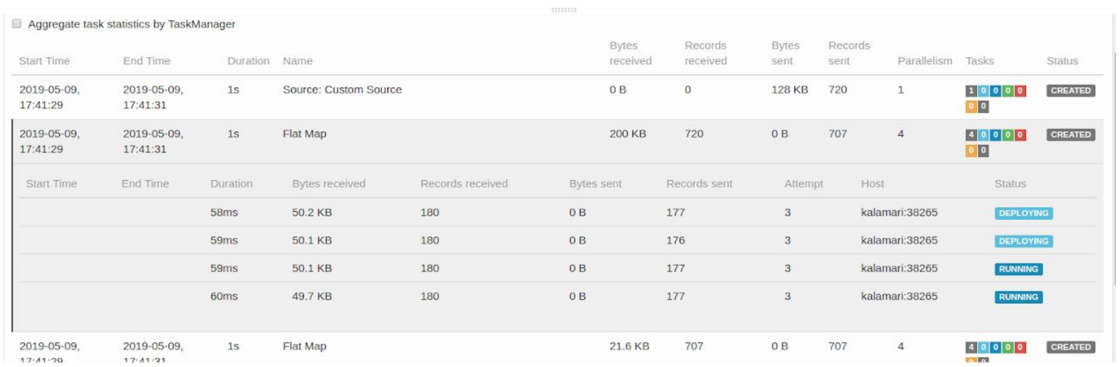
Even though there are now enough slots to restart the failed application, Flink will not do so. If you check the failed job's configuration, you will see that we had not set the value of "Max number of execution retries".

Tasks

#1: Add the following lines to your analysis program, rebuild, and resubmit the jar.

```
// Set a fixed delay restart strategy with a maximum of 5 restart attempts
// and a 1s interval between retries
env.setRestartStrategy(RestartStrategies.fixedDelayRestart(5, 1000));
```

What happens if you now kill and restart the TaskManager? Can you see your job re-starting like below?



☐ Aggregate task statistics by TaskManager

Start Time	End Time	Duration	Name	Bytes received	Records received	Bytes sent	Records sent	Parallelism	Tasks	Status
2019-05-09, 17:41:29	2019-05-09, 17:41:31	1s	Source: Custom Source	0 B	0	128 KB	720	1	1 0 0 0 0 0 0	CREATED
2019-05-09, 17:41:29	2019-05-09, 17:41:31	1s	Flat Map	200 KB	720	0 B	707	4	4 0 0 0 0 0 0	CREATED
2019-05-09,	2019-05-09,	1s	Flat Map	21.6 KB	707	0 B	707	4	4 0 0 0 0	CREATED

Start Time	End Time	Duration	Bytes received	Records received	Bytes sent	Records sent	Attempt	Host	Status
		58ms	50.2 KB	180	0 B	177	3	kalamari:38265	DEPLOYING
		59ms	50.1 KB	180	0 B	176	3	kalamari:38265	DEPLOYING
		59ms	50.1 KB	180	0 B	177	3	kalamari:38265	RUNNING
		60ms	49.7 KB	180	0 B	177	3	kalamari:38265	RUNNING

#2: Execute the failure and restart experiment and output results to a file so you can compare results across restarts (**Hint:** make sure the file name per attempt is unique, otherwise your job will fail with a `java.io.IOException`). Since we are not using managed state and checkpointing is not enabled, your application will start reading from the beginning of the Kafka topic every time it restarts. Can you verify this behavior by looking at the output?

2. Use managed state and checkpoints

By setting the restart strategy, we have now instructed Flink to automatically restart a failed application when resources are available. However, state will be lost and be recreated from scratch after every restart. To avoid losing our work, we can enable checkpointing as follows:

```
// Take a checkpoint every 10s
env.enableCheckpointing(10000);
```

Tasks

#1: Consult the Flink docs and change the `ComputeDiffs` function to use managed state. Which primitive is appropriate to use? Why? Note that there is a `keyBy()` call before the flatmap and the state you will define is thus scoped to a particular key.

#2: The default in-memory state backend might be fine for the purposes of this exercise but it will cause problems if you use it in an application with large state. Refer to the Flink documentation and change the state backend to RocksDB. Let the application trigger 4-5 checkpoints and note down their size and duration. You can monitor checkpoint metrics via the web interface: http://localhost:8081/

#3. Find the configuration property that enables incremental checkpointing. Enable it, re-run your application, and note down the checkpoint sizes and durations. How do they compare with the non-incremental case?

3. Re-configure the application from a savepoint

While periodic checkpoints ensure that your application state will not be lost in the case of failure, the mechanism they rely on can also be used for re-configuration. A *Savepoint* is an on-demand checkpoint of a streaming application's state and can be used to re-start the same or a slightly modified version of your application from a consistent point.

For the purpose of this assignment, we will only try making two small changes to the wikipedia analysis application. In order to use savepoints, you first need to assign unique ids to your operators with the `uid()` method as follows:

```
DataStream<String> stream = env.
  // Stateful source (e.g. Kafka) with ID
  .addSource(new StatefulSource())
  .uid("source-id") // ID for the source operator
  .shuffle()
  // Stateful mapper with ID
  .map(new StatefulMapper())
  .uid("mapper-id"); // ID for the mapper
```

Next, open the `flink-conf.yaml` file and set the `state.savepoints` property to the directory where you want the savepoint to be stored.

<u>Tasks</u>
#1: Rebuild the wikipedia analysis application and after a few minutes use the command-line interface to cancel the job with a savepoint. Restore your job from the savepoint using different parallelism for the `ComputeDiffs` operator. Consult the web interface to ensure your change took effect.

#2: Savepoints are also useful for code updates, e.g. to fix bugs or even change business logic. Update the code in your `ComputeDiffs` operator to output diffs in KB instead of bytes. What happens when you re-store from a savepoint? Are the counters correct?