Google Summer of Code

Introduction

This area is dedicated to Google Summer Of Code projects past and present.
Primarily it should harvest documentation relating to these projects but most importantly it should act as a platform for anyone to contribute towards Samza GSoC project initiatives.

2015

Proposal: Expanding Apache Samza’s consumers/producers to a wider spectrum of messaging systems

Description

Different JIRA issues were logged stating that Samza should be able to consumer and produce data to other distributed publisher/subscriber systems e.g. Amazon Kinesis, Apache ActiveMQ, Twitter's Kreskel, and other systems. The community proposed that the integration with such systems could be done as a the Google Summer of Code project for 2015 .

Student

Renato Marroquin

Mentor

Yan Fang

Full proposal

Proposal Title : Expanding Apache Samza's consumers/producers to a wider spectrum of messaging systems
Student Name: Renato Marroquin
Student Email : renatoj.marroquin@gmail.com
Project Description: Apache Samza[1] is a distributed stream-processing framework that can be deployed on top of Apache Hadoop Yarn in order to leverage existing Hadoop infrastructure. Samza's main messaging system is Apache Kafka, but in spite of the fact that Samza has been developed around Kafka's architecture, it has become a very popular stream processing system. In order to help Samza grow even more, the motivation of this project is to add the ability to read/write data from/to different message queues. This project aims to expand the capabilities of Samza for consuming/producing messages to two popular message queues, Apache ActiveMQ[2] (SAMZA-587) and Amazon Kinesis[3] (SAMZA-489). ActiveMQ is a message broker implementing Java Message Service API 1.1. It supports clients in many different programming languages e.g. C, C++, Erlang, among many others. Amazon Kinesis provides a cloud-based service for data processing over distributed data streams. It supports clients in Java, Python, and Ruby. Samza has an example application that uses Kafka as its messaging system. This sample application can be used for testing the resulting integration and also for providing users the ability to specify different messaging systems for testing Samza.
Tentative project architecture: Apache Samza has a well-defined API for new system consumers and producers in [4]. Therefore, each of the new messaging systems will extend the SystemProducer and SystemConsumer interfaces. The integration with Apache ActiveMQ will reside in a separate maven module similar to the "samza-kafka" module. The integration with Amazon Kinesis will be an external module that will reside on an external repository. This is because a project's committer, and an Amazon engineer have started Amazon Kinesis integration with Samza. Although the code will not reside on the project's main code repository, this integration will help on growing the community for Samza because of Amazon Kinesis popularity. Both integrations will have to be modelled similarly to the way the Kafka consumer/producer have been implemented. The three systems, ActiveMQ, Kinesis, and Kafka, provide similar functionality, however there exist differences that will need to be studied and understood to write a successful integration. For example, the way they deserialize/serialize data to be read/written, the way they handle offsets internally or the API each system offers.

Timeline:

Before 25 May. Getting familiarized with the Amazon Kinesis and Apache ActiveMQ ways of reading and writing data.
27 April - 24 May. Getting familiarized with the Apache Samza code base. Helping in generating documentation for the project, writing unit tests for it, or solving JIRA issues important to the Samza community.
25 May - 26 June. Complete the integration with Amazon Kinesis by implementing KinesisProducer, KinesisConsumer classes and their respective configuration classes for its correct functioning.
27 June – 27 July. Complete the integration with Amazon Kinesis by implementing ActiveMQProducer, ActiveMQConsumer classes and their respective configuration classes for its correct functioning.
28 July - 10 August. Profile the applications for further improvement while using Samza's sample application as an end-to-end integration test between the projects.
11 August - 16 August. Write required tests and documentation for both integrations.
17 August - 21 August. Preparing for final release. Refactoring. Adding missing documentation for users.

JIRA Issue

Contribute with some JIRA issues to get more confortable with the code base.

https://issues.apache.org/jira/browse/SAMZA-587
https://issues.apache.org/jira/browse/SAMZA-489

Project Deliverables

- Kinesis-Samza consumer and producer systems
- ActiveMQ-Samza consumer and producer systems
- JUnit Test cases for integrated systems

Implementation Approach

Phase 1

Task	Status
• Implement sample application based on Samza HelloWorld application.	DONE
• Try out Amazon Kinesis basic functionality.	DONE
• Implement basic prototype of SystemConsumer and SystemProducer API for Kinesis integration.	DONE
• Improve prototype of Kinesis integration.	IN-PROGRESS

Phase 2

Task	Status

• Try out Apache ActiveMQ basic functionality.	TODO
• Implement basic prototype of SystemConsumer and SystemProducer API for ActiveMQ integration	TODO
• Optimize prototype of ActiveMQ integration.	TODO

Phase 3

Task	Status
• Write tests for Kinesis integration.	TODO
• Write tests for ActiveMQ integration.	TODO

Phase 4

Task	Status
• Write documentation for new integrated systems.	TODO
• Review checkpointing mechanism for new integrated systems.	TODO

Project Reports

Report Ending Week June 22nd

Project description

Integration between Apache Samza and Amazon Kinesis

Review of Previous Actions

Discussion about current implementation (pros and cons) which is based on the Amazon Kinesis Client Library (KCL).

For this first phase of the project two approaches were explored:

- An approach was based on creating a single KCL for each Yarn Container. Although this would use less resources and allow the creation of more containers within the same server, it also suffers from the fact that the ingested messages are mapped to specific tasks.
- The other approach allowed every task to have its own KCL. This means that each task could handle the number of shard readers the KCL decides it has to use. This is based on the number of available shards and workers registered through the KCL).

In any of these approaches, loosing messages was a possibility in the cases that are explained below.

Objectives

We needed to review the current implementation as there were specific cases in which pounds on higher scalability and robustness through the usage of KCL. Although this could lead to lost of messages in the Samza side.

Pros

- The main advantage of the above explained approaches is that we can rely on the KCL capability of load-balancing, managing resharding, error handling, and request retrying.
- The current implementation provides a KCL to each task. If any of the shards assigned to a specific container would be deleted or merged, then the task's KCLs would rebalance their work by coordinating through Amazon DynamoDB and know where to fetch the data from automatically. All this happens behinds the curtains, and Samza doesn't have to be bother with it.

Cons

1. **If a whole container goes down**
    - The tasks assigned to the container will also go down.

    - The tasks' KCLs would coordinate among them and start consuming from the (temporary) unconsumed shards. This behaviour is great, but the problem is that the messages from the (temporary) unconsumed shards haven't been assigned to this still alive container.
    - This in turn means that the tasks from that container will not process such messages.
    - Even if the dead container comes back, the tasks' KCLs will have already "read" messages from the unconsumed data, and will start from a later point, skipping the messages read by the tasks from the container that didn't go down.
2. If a task goes down
    - Then the shards assigned to it through its KCL would be redirected to the any other KCL worker. This means that if we are lucky then another task (with its KCL) inside the same container would get it, and everything would just magically work.
    - If the shards assigned to the dead task are redirected to a different container, then the fail case is the same as if the whole container would have been down.
3. If resharding happens (adding new Kinesis shards or deleting existing ones)
    - Then the KCLs within the tasks will take care of this. But again the messages might end up in containers where no-tasks were assigned to them.

Future Actions

My mentor and me decided that it is better to have a fully correct implementation first, rather than having an implementation with more options but that could present undesired behaviour.

Change the usage of the Kinesis Client library over the usage of a finer grain Kinesis access method. This is based on using the simple consumer request provided by Amazon. This would help us in two important aspects of the work:

- The correctness of the implementation (really important one).
- The ability to control checkpointing from the Samza side. We have to do checkpointing from the Kinesis side as well, but in case of failures, Samza should be able to handle them at least with the current implementation. Maybe at some point the checkpointing can also be given to the underlaying system.

I got the chance to discuss some of these ideas also with some people from Amazon, and this issue (the hard mapping between containers/tasks/partitions /messages) is something they are aware of the initial implementation.

Some ideas for a future improvement of this part is:

- Creating a different hashing strategy from the incoming messages to the tasks. This has to be agnostic of the number of partitions, and only has to rely on the messages and on the current tasks being executing.
- Thus, we would probably have to create a different ContainerModel (so we don't tight the partitions to the tasks) and a different MessageChooser (so we can redirect new messages to specific tasks)

I will start documenting the follow up JIRA issues that can follow this initial integration because the previous ideas fall out of the scope of this GSoC project.

Mentors Comments