Apache Samza

AJIT BALAGA, S17-IR-2004[1]

[1] *School of Informatics and Computing, Bloomington, IN 47408, U.S.A.*
[*] *Corresponding authors: abalaga@iu.edu, ajit.balaga@gmail.com*

project-000, February 27, 2017

Apache Samza is a stream processing framework which enables usesrs to analyze streaming datawith ease. The concepts required streaming data and the architecture behind Samza is presented here. Samza relies on Kafka and YARN, and its relationship with the two technologies is elaborated upon. The API is introduced as a precursor to future streaming projects.

Keywords: Cloud, I524

https://github.com/argetlam115/sp17-i524/blob/master/paper1/S17-IR-2004/report.pdf

INTRODUCTION

Messaging systems are a method of implementing realtime asynchronous computation, where messages can be added to a message queue, pub-sub system, or log aggregation system whenever an event occurs. Consumers or subscribers read these messages from the queues or systems and take actions based on the message contents. A messaging system stores messages and waits for consumers to consume them. Samza, a stream processing system, is a higher level of abstraction on top of messaging systems.[1]

SAMZA

Samza uses YARN and Kafka to provide a framework for stage-wise stream processing and partitioning. Samza is a stream processing framework with the following features:[1]

- Simple API
- Managed state
- Fault tolerance
- Durability
- Scalability
- Pluggable
- Processor isolation

The Samza client uses YARN to run a Samza job: YARN starts and supervises one or more SamzaContainers, and your processing code runs inside those containers. The input and output for the Samza StreamTasks come from Kafka brokers that are co-located on the same machines as the YARN NMs.[2] Samza is made up of three layers:

- A streaming layer
- An execution layer
- A processing layer

Samza provides out of the box support for all three layers.

- Streaming: Kafka
- Execution: YARN
- Processing: Samza API

Samza's execution and streaming layers are differentiable from the processing layer allowing developers to implement alternatives of their choice.

STREAMING WITH KAFKA

Kafka is a distributed pub/sub and message queueing system that provides at-least once messaging guarantees, and highly available partitions. Each topic (streams in Kafka) is partitioned and replicated across multiple machines called brokers. When a producer sends a message to a topic, it provides a key, which is used to determine which partition the message should be sent to. The Kafka brokers receive and store the messages that the producer sends. Kafka consumers can then read from a topic by subscribing to messages on all partitions of a topic.[3] Kafka has some interesting properties:

- All messages with the same key are guaranteed to be in the same topic partition.
- A topic partition is a sequence of messages in order of arrival and are stored using a monotonically increasing offset. The consumer can keep track of the offset by storing the offset of the last message it has processed.

RESOURCE MANAGEMENT WITH YARN

YARN is Hadoop's next-generation cluster scheduler which allows you to allocate a number of containers in a cluster of machines, and execute arbitrary commands on them. Samza uses YARN to manage deployment, fault tolerance, logging, resource isolation, security, and locality. YARN has three important pieces: a ResourceManager, a NodeManager, and an ApplicationMaster. In a YARN grid, every machine runs a NodeManager, which is responsible for launching processes on that machine. A ResourceManager talks to all of the NodeManagers to tell them what to run. Applications talk to the ResourceManager when they wish to run something on the cluster. The third piece, the ApplicationMaster, is actually application-specific code that runs in the YARN cluster. It's responsible for managing the application's workload, asking for containers, and handling notifications when one of its containers fails.[4] Samza provides a YARN ApplicationMaster and a YARN job runner out of the box.The Samza client talks to the YARN RM when it wants to start a new Samza job. The YARN RM talks to a YARN NM to allocate space on the cluster for Samza's ApplicationMaster. Once the NM allocates space, it starts the Samza AM. After the Samza AM starts, it asks the YARN RM for one or more YARN containers to run SamzaContainers. Again, the RM works with NMs to allocate space for the containers. Once the space has been allocated, the NMs start the Samza containers.[5]

RELATED CONCEPTS

Streams

A stream is composed of immutable messages of a similar type or category. Messages can be appended to a stream or read from a stream. A stream can have any number of consumers, and reading from a stream doesn't delete the message. Messages can optionally have an associated key which is used for partitioning. [1]

Jobs

A Samza job performs a logical transformation on a set of input streams to append output messages to set of output streams. In order to scale the throughput of the stream processor, streams and jobs are cut into smaller units of parallelism: partitions and tasks.[1]

Partitions

Each stream is broken into a number of partitions, each with a few properties. Each partition is an ordered sequence of messages with an identifier called the offset, unique to its partition. When a message is appended to a stream, it is appended to only one of the stream's partitions.[1]

Tasks

A job is broken into multiple tasks. Each task consumes data from a partition. A task processes messages from each of its input partitions sequentially, in the order of message offset, in parallel for all partitions available. The YARN scheduler assigns each task to a machine, so the job as a whole can be distributed across many machines. The number of tasks in a job is determined by the number of input partitions (there cannot be more tasks than input partitions, or there would be some tasks with no input). However, you can change the computational resources assigned to the job (the amount of memory, number of CPU cores, etc.) to satisfy the job's needs. See notes on containers below. The assignment of partitions to tasks never changes: if a task is on a machine that fails, the task is restarted elsewhere, still consuming the same stream partitions.[1][6]

Dataflow Graphs

We can compose multiple jobs to create a dataflow graph, where the edges are streams containing data, and the nodes are jobs performing transformations. This composition is done purely through the streams the jobs take as input and output. The jobs are otherwise totally decoupled: they need not be implemented in the same code base, and adding, removing, or restarting a downstream job will not impact an upstream job. These graphs are often acyclic—that is, data usually doesn't flow from a job, through other jobs, back to itself. However, it is possible to create cyclic graphs if you need to.[1]

Containers

Partitions and tasks are both logical units of parallelism—they don't correspond to any particular assignment of computational resources. Containers are the unit of physical parallelism, and a container is essentially a Unix process. Each container runs one or more tasks. The number of tasks is determined automatically from the number of partitions in the input and is fixed, but the number of containers is specified by the user at run time and can be changed at any time.[1]

API

When writing a stream processor for Samza, you must implement the StreamTask interface. When you run your job, Samza will create several instances of your class. These task instances process the messages in the input streams. For each message that Samza receives from the task's input streams, the process method is called. The envelope contains three things of importance: the message, the key, and the stream that the message came from. The key and value are declared as Object, and need to be cast to the correct type. If you don't configure a serializer/deserializer, they are typically Java byte arrays. A deserializer can convert these bytes into any other type.[1] The getSystemStreamPartition() method returns a SystemStreamPartition object, which tells you where the message came from. It consists of three parts:

- The system: the name of the system from which the message came, as defined in your job configuration. You can have multiple systems for input and/or output, each with a different name.
- The stream name: the name of the stream (topic, queue) within the source system. This is also defined in the job configuration.
- The partition: a stream is normally split into several partitions, and each partition is assigned to one StreamTask instance by Samza.

The API looks like this:

```
/** A triple of system name, stream name and partition. */
public class SystemStreamPartition extends SystemStream
/** The name of the system which provides this stream. It is
defined in the Samza job's configuration. */
public String getSystem() ...
/** The name of the stream/topic/queue within the system. */
```

```
public String getStream() ...
/** The partition within the stream. */
public Partition getPartition() ...
```

CONCLUSION

Samza is a very powerful tool to work on streaming data. With its simple approach, it allows us to analyze large amounts of streaming data on the go. The architecture allows the developer to utilize their own resource manager and their message handling system. Samza's architecture is very similar to hadoop, enabling users to get started with their applications quickly and making the learning curve shallow. With its simpleapi, Samza is a comfortable technology for analyzing streaming data.

REFERENCES

[1] Web Page. [Online]. Available: http://samza.apache.org

[2] T. Feng, Z. Zhuang, Y. Pan, and H. Ramachandra, "A memory capacity model for high performing data-filtering applications in samza framework," in *Big Data (Big Data), 2015 IEEE International Conference on*. IEEE, 2015, pp. 2600–2605.

[3] M. Kleppmann and J. Kreps, "Kafka, samza and the unix philosophy of distributed data," *Bulletin of the IEEE CS Technical Committee on Data Engineering*, 2015.

[4] G. Wang, J. Koshy, S. Subramanian, K. Paramasivam, M. Zadeh, N. Narkhede, J. Rao, J. Kreps, and J. Stein, "Building a replicated logging system with apache kafka," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1654–1655, 2015.

[5] B. Srikanth and V. K. Reddy, "Efficiency of stream processing engines for processing bigdata streams," *Indian Journal of Science and Technology*, vol. 9, no. 14, 2016.

[6] J. Samosir, M. Indrawan-Santiago, and P. D. Haghighi, "An evaluation of data stream processing systems for data driven applications," *Procedia Computer Science*, vol. 80, pp. 439–449, 2016.