Architecture

These three pieces fit together to form Samza:

YARN

[YARN](http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html) (Yet Another Resource Negotiator) is Hadoop’s next-generation cluster scheduler. It allows you to allocate a number of *containers* (processes) in a cluster of machines, and execute arbitrary commands on them.

When an application interacts with YARN, it looks something like this:

1. **Application**: I want to run command X on two machines with 512MB memory.
2. **YARN**: Cool, where’s your code?
3. **Application**: http://path.to.host/jobs/download/my.tgz
4. **YARN**: I’m running your job on node-1.grid and node-2.grid.

Samza uses YARN to manage deployment, fault tolerance, logging, resource isolation, security, and locality. A brief overview of YARN is below; see [this page from Hortonworks](http://hortonworks.com/blog/apache-hadoop-yarn-background-and-an-overview/) for a much better overview.

YARN Architecture

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, in turn, 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 (usually UNIX processes), and handling notifications when one of its containers fails.

Samza and YARN

Samza provides a YARN ApplicationMaster and a YARN job runner out of the box. The integration between Samza and YARN is outlined in the following diagram (different colors indicate different host machines):

**(figure 1 3rows)**

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](http://samza.apache.org/learn/documentation/latest/container/samza-container.html). Again, the RM works with NMs to allocate space for the containers. Once the space has been allocated, the NMs start the Samza containers.

Samza

Samza uses YARN and Kafka to provide a framework for stage-wise stream processing and partitioning. Everything, put together, looks like this (different colors indicate different host machines):

**(figure 2 4rows)**

The Samza client uses YARN to run a Samza job: YARN starts and supervises one or more[SamzaContainers](http://samza.apache.org/learn/documentation/latest/container/samza-container.html), and your processing code (using the [StreamTask](http://samza.apache.org/learn/documentation/latest/api/overview.html) API) runs inside those containers. The input and output for the Samza StreamTasks come from Kafka brokers that are (usually) co-located on the same machines as the YARN NMs.

Example

Let’s take a look at a real example: suppose we want to count the number of page views. In SQL, you would write something like:

SELECT user\_id, COUNT(\*) FROM PageViewEvent GROUP BY user\_id

Although Samza doesn’t support SQL right now, the idea is the same. Two jobs are required to calculate this query: one to group messages by user ID, and the other to do the counting.

In the first job, the grouping is done by sending all messages with the same user ID to the same partition of an intermediate topic. You can do this by using the user ID as key of the messages that are emitted by the first job, and this key is mapped to one of the intermediate topic’s partitions (usually by taking a hash of the key mod the number of partitions). The second job consumes the intermediate topic. Each task in the second job consumes one partition of the intermediate topic, i.e. all the messages for a subset of user IDs. The task has a counter for each user ID in its partition, and the appropriate counter is incremented every time the task receives a message with a particular user ID.

**(figure3 group by userid)**

If you are familiar with Hadoop, you may recognize this as a Map/Reduce operation, where each record is associated with a particular key in the mappers, records with the same key are grouped together by the framework, and then counted in the reduce step. The difference between Hadoop and Samza is that Hadoop operates on a fixed input, whereas Samza works with unbounded streams of data.

Kafka takes the messages emitted by the first job and buffers them on disk, distributed across multiple machines. This helps make the system fault-tolerant: if one machine fails, no messages are lost, because they have been replicated to other machines. And if the second job goes slow or stops consuming messages for any reason, the first job is unaffected: the disk buffer can absorb the backlog of messages from the first job until the second job catches up again.

By partitioning topics, and by breaking a stream process down into jobs and parallel tasks that run on multiple machines, Samza scales to streams with very high message throughput. By using YARN and Kafka, Samza achieves fault-tolerance: if a process or machine fails, it is automatically restarted on another machine and continues processing messages from the point where it left off.

API Overview

When writing a stream processor for Samza, you must implement the [StreamTask](http://samza.apache.org/learn/documentation/latest/api/javadocs/org/apache/samza/task/StreamTask.html) interface:

public class MyTaskClass implements StreamTask {

public void process(IncomingMessageEnvelope envelope,

MessageCollector collector,

TaskCoordinator coordinator) {

// process message

When you run your job, Samza will create several instances of your class (potentially on multiple machines). These task instances process the messages in the input streams.

In your job’s configuration you can tell Samza which streams you want to consume. An incomplete example could look like this (see the [configuration documentation](http://samza.apache.org/learn/documentation/latest/jobs/configuration.html) for more detail):

# This is the class above, which Samza will instantiate when the job is run

task.class=com.example.samza.MyTaskClass

# Define a system called "kafka" (you can give it any name, and you can define

# multiple systems if you want to process messages from different sources) systems.kafka.samza.factory=org.apache.samza.system.kafka.KafkaSystemFactory

# The job consumes a topic called "PageViewEvent" from the "kafka" system

task.inputs=kafka.PageViewEvent

# Define a serializer/deserializer called "json" which parses JSON messages

serializers.registry.json.class=org.apache.samza.serializers.JsonSerdeFactory

# Use the "json" serializer for messages in the "PageViewEvent" topic

systems.kafka.streams.PageViewEvent.samza.msg.serde=json

For each message that Samza receives from the task’s input streams, the *process* method is called. The [envelope](http://samza.apache.org/learn/documentation/latest/api/javadocs/org/apache/samza/system/IncomingMessageEnvelope.html) contains three things of importance: the message, the key, and the stream that the message came from.

/\*\* Every message that is delivered to a StreamTask is wrapped

\* in an IncomingMessageEnvelope, which contains metadata about

\* the origin of the message. \*/

public class IncomingMessageEnvelope {

/\*\* A deserialized message. \*/

Object getMessage() { ... }

/\*\* A deserialized key. \*/

Object getKey() { ... }

/\*\* The stream and partition that this message came from. \*/

SystemStreamPartition getSystemStreamPartition() { ... }

}

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](http://samza.apache.org/learn/documentation/latest/container/serialization.html), they are typically Java byte arrays. A deserializer can convert these bytes into any other type, for example the JSON deserializer mentioned above parses the byte array into java.util.Map, java.util.List and String objects.

The getSystemStreamPartition() method returns a [SystemStreamPartition](http://samza.apache.org/learn/documentation/latest/api/javadocs/org/apache/samza/system/SystemStreamPartition.html) object, which tells you where the message came from. It consists of three parts:

1. 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.
2. The *stream name*: the name of the stream (topic, queue) within the source system. This is also defined in the job configuration.
3. The [*partition*](http://samza.apache.org/learn/documentation/latest/api/javadocs/org/apache/samza/Partition.html): 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() { ... }

}

In the example job configuration above, the system name is “kafka”, the stream name is “PageViewEvent”. (The name “kafka” isn’t special — you can give your system any name you want.) If you have several input streams feeding into your StreamTask, you can use the SystemStreamPartition to determine what kind of message you’ve received.

What about sending messages? If you take a look at the process() method in StreamTask, you’ll see that you get a [MessageCollector](http://samza.apache.org/learn/documentation/latest/api/javadocs/org/apache/samza/task/MessageCollector.html).

/\*\* When a task wishes to send a message, it uses this interface. \*/

public interface MessageCollector {

void send(OutgoingMessageEnvelope envelope);

}

To send a message, you create an [OutgoingMessageEnvelope](http://samza.apache.org/learn/documentation/latest/api/javadocs/org/apache/samza/system/OutgoingMessageEnvelope.html) object and pass it to the message collector. At a minimum, the envelope specifies the message you want to send, and the system and stream name to send it to. Optionally you can specify the partitioning key and other parameters. See the[javadoc](http://samza.apache.org/learn/documentation/latest/api/javadocs/org/apache/samza/system/OutgoingMessageEnvelope.html) for details.

**NOTE:** Please only use the MessageCollector object within the process() method. If you hold on to a MessageCollector instance and use it again later, your messages may not be sent correctly.

For example, here’s a simple task that splits each input message into words, and emits each word as a separate message:

public class SplitStringIntoWords implements StreamTask {

// Send outgoing messages to a stream called "words" in the "kafka" system.

private final SystemStream OUTPUT\_STREAM =

new SystemStream("kafka", "words");

public void process(IncomingMessageEnvelope envelope, MessageCollector collector,

TaskCoordinator coordinator) {

String message = (String) envelope.getMessage();

for (String word : message.split(" ")) {

// Use the word as the key, and 1 as the value.

// A second task can add the 1's to get the word count.

collector.send(new OutgoingMessageEnvelope(OUTPUT\_STREAM, word, 1));

}

}

}

Security

Samza provides no security. All security is implemented in the stream system, or in the environment that Samza containers run.

Securing Streaming Systems

Samza does not provide any security at the stream system level. It is up to individual streaming systems to enforce their own security. If a stream system requires usernames and passwords in order to consume from specific streams, these values must be supplied via configuration, and used at the StreamConsumer/StreamConsumerFactory implementation. The same holds true if the streaming system uses SSL certificates or Kerberos. The environment in which Samza runs must provide the appropriate certificate or Kerberos ticket, and the StreamConsumer must be implemented to use these certificates or tickets.

Securing Kafka

Kafka provides no security for its topics, and therefore Samza doesn’t provide any security when using Kafka topics.

Securing Samza’s Environment

The most important thing to keep in mind when securing an environment that Samza containers run in is that **Samza containers execute arbitrary user code**. They must considered an adversarial application, and the environment must be locked down accordingly.