A Simple Stream Processing System - Akka

Networked Software for Distributed Systems

Matteo Mormile, 10666565

1. Introduction

The project is based on the construction of a simple stream processing system using actor model. The system is made up of a series of operators that use mathematical operators to gradually alter our data flow. Each operator in the pipeline is defined using a window `(size, slide)` and an aggregation function to process values in the current window, and each message passing through the operators contains a pair `<key, value>`. Upon completion of the window, the operator calculates the total value and transfers the outcome to the subsequent operator. The slide indicates how much data needs to be received before the operation is applied again. There are numerous independent streams in our system stream processing where the same kind of operators may appear repeatedly but in different instances.
For example, consider a pipeline to process sensor data, keys represent the type of data (temperature and humidity) and values are the actual sensor reading. Aggregation functions are operators like average, max and are applied successively to the data produced by the sensor.

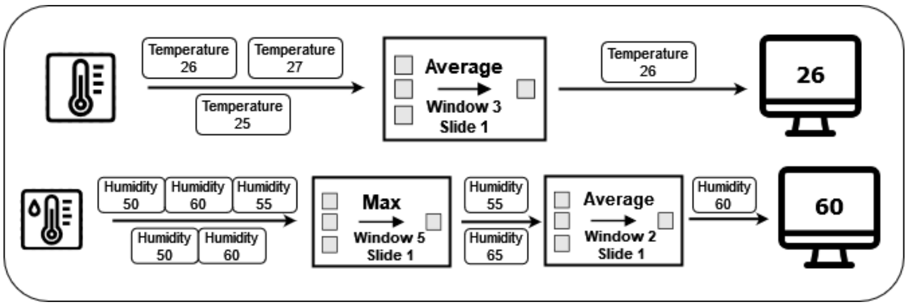


Figure 1: Example of how a parallel stream processing works.

2. Architecture

The application is built by using Akka, a set of libraries for designing scalable, resilient systems. The project is based on a client-server paradigm, in which individual sensors `(Client)` continue to send data to the `Server` for analysis. In the server there will be operator pipelines, represented as a set of actor lists, ready to receive data. In this way, we will have one manager who will be responsible for creating the actors that generate the data and other manager who will be responsible for creating the pipeline of operators. To enable communication between actors that produce data and actors that perform operations on it, I will set up a cluster of Kafka nodes with both types of actors.

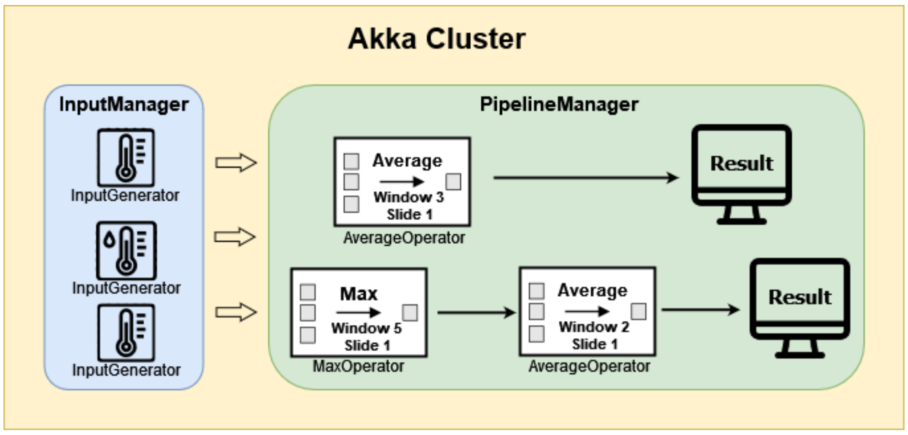


Figure 2: Client - Server architecture of the project.

2.1. Input

The part of the project that is responsible for generating the data and sending it to the pipeline consists of the following classes:

- **InputMain**: after creating the actor System according to the specification in the application.conf file and the arguments, it creates the InputManager actor that will take the role of supervisor over all InputGenerator(s) created. The main process will then continue, depending on the commands read via cli, to talk to the input manager creating new InputGenerator(s) generators.
- **InputManager**: supervisor of the actors that generate the data to be sent to the pipeline. The InputManager actor is also in charge of talking to the PipelineManager and receiving the reference to the first actor in the pipeline to which the corresponding InputGenerator is to send the data.
- **InputGenerator**: actor controlled by the InputManager. Is responsible for creating values at regular intervals and sending them to the pipeline for processing.

2.2. Pipeline

As in the part that deals with input, we also find a main that will be in charge of creating a pipeline manager through the actor System. In this case, however, the main process will wait for any input to generate false malfunctions in the pipeline. The actors that we find in the pipeline are:

- **PipelineManager**: is responsible for supervising and creating pipelines. Pipelines are lists of actors. The pipeline manager receives a message from the input manager containing the details of the pipeline to be created, and after instantiating the actors returns the address of the first operator in the pipeline to the input manager.

```
List<Operator> operators = new ArrayList<>(msg.getOperatorList());
for (int i=0;i<operators.size();i++) {
    ActorRef next = createOperator(operators.get(i).getOperationType(), msg.getKey());
    if (prev != null) {
        prev.tell(new Config(next, msg.getKey(), operators.get(i-1).getWindowSize(),
                     operators.get(i-1).getSlideSize()), self());
    } else
        start = next;
    prev = next;
}
if (prev != null) {
    prev.tell(new Config(null, msg.getKey(), operators.get(/*last*/).getWindowSize(),
                operators.get(/*last*/.getSlideSize()), self());
}
ReturnPipeline resp = new ReturnPipeline(start, msg.getKey());
sender().tell(resp, self());
```

Listing 1: Create list of operator actors and link them

When the operator completes its window it calculates the result and, if the pointer to the next actor is not null, forwards the result to the next actor, otherwise it prints the result. The pipeline manager also takes care of forwarding any kill message to the right actor operator to simulate a failure.

- **AbstractOperator**: is the abstract class that defines an operator. In the abstract class the functions common to each operator such as receiving messages and forwarding them are defined. Instead, the function to be applied to the elements collected in the window is overridden by the actual operator classes. The size of the window and the slide are indicated after the actor is created by a configuration message.

```
@Override
protected double performOperation(List<Double> values) {
    double sum = 0.0;
    for (Double value : values) {
        sum += value;
    }
    return sum / values.size();
}
```

Listing 2: Override method to apply when window is full

2.3. Messages

We find several messages exchanged between actors in our project, for simplicity the actors between which they are exchanged are indicated.

- **InputMain** $\longrightarrow$ **InputManager** $\longrightarrow$ **PipelineManager**:
    - *CreatePipeline*: contains the list of operators that will compose the pipeline and the key that will represent the data stream.
- **InputMain** $\longleftarrow$ **InputManager** $\longleftarrow$ **PipelineManager**:
    - *ReturnPipeline*: contains the reference to the first actor (operator) in the pipeline to which the input generator will send data.
- **InputMain** $\longrightarrow$ **InputGenerator**:
    - *StartPipeline*: contains the address of the first pipeline operator. After receiving this message, the input generator will start sending messages at regular intervals.
- **PipelineManager** $\longrightarrow$ **Operator**:
    - *Config*: contains the configuration values of each operator such as: the window size, slide, the key of the values to be analyzed and the reference to the next operator in the pipeline.
- **InputGenerator** $\longrightarrow$ **Operator** $\longrightarrow$ **Operator**:
    - *KeyValueMessage*: message containing the values to be processed. Every time that the operator performs an operation, a new message is created with the same key (it identifies the stream) and the result obtained.

- **PipelineMain** $\longrightarrow$ **PipelineManager** $\longrightarrow$ **Operator**:
  - *KillActor*: message containing the name of the operator to be killed. The pipeline manager, after retrieving the operator reference, will forward the message. By not recognizing the message type, the operator will raise an exception.

3. Testing and Conclusions

In order to test the application with the pipeline and inputs running on the same machine we can simply run our .jar files. Instead, if we want to run the single components on two different machines, we need to specify through the main arguments the IP addresses of the machines.

```
java -jar Pipeline.jar <SERVER_IP>
java -jar Pipeline.jar 192.168.1.55

java -jar Input.jar <CLIENT_IP> <SERVER_IP>
java -jar Input.jar 192.168.1.66 192.168.1.55
```

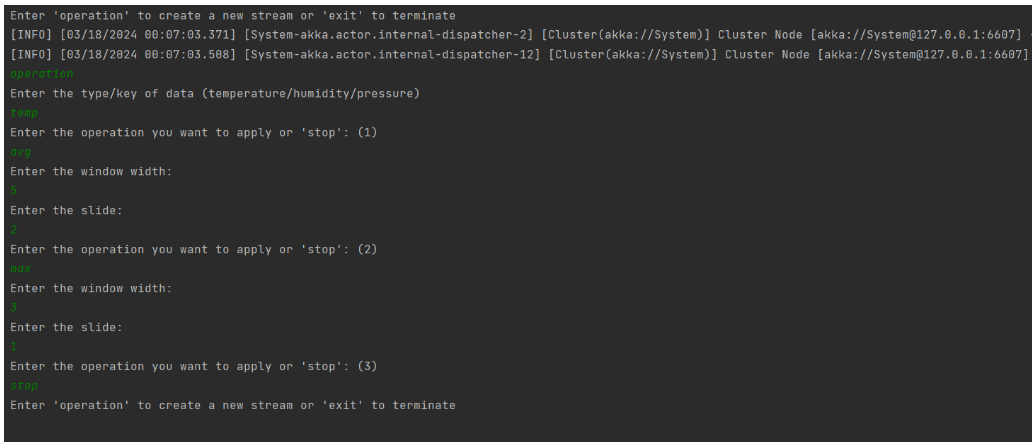
Listing 3: Commands used to run the application in a distributed environment

The InputMain and PipelineMain files will use these parameters to modify our system configuration file after reading it.

```
// local or distributed mode
if (args.length < 1) {
  System.out.println("IP address needed to be able to publish server <SERVER_IP>.");
  System.out.println("System will start in local mode!");
}else{
  // set server properties
  String serverIP = args[0];
  System.out.printf("Distributed mode activated: server (%s)\n",serverIP);
  config = config.withValue("akka.cluster.seed-nodes", [...]);
  config = config.withValue("akka.remote.artery.canonical.hostname", [...]);
}
```

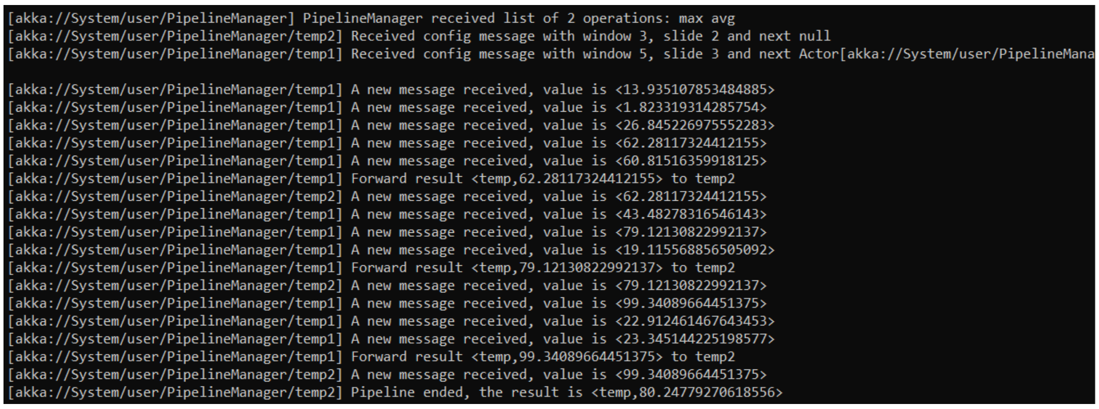
Listing 4: Before creating the System actor, PipelineMain modifies the configuration values

To test if everything was working correctly, we tested the application first locally, using multiple processes on the same machine, and then in a distributed environment connecting multiple machines in the same network. In both scenarios everything worked as expected allowing every data in the pipelines to be processed. Thanks to the supervisor strategy implemented in the pipeline manager, operator crashes are totally managed and can only be noticed by looking at the central server log. Here there are some pictures of a sample test:



```
Enter 'operation' to create a new stream or 'exit' to terminate
[INFO] [03/18/2024 00:07:03.371] [System-akka.actor.internal-dispatcher-2] [Cluster(akka://System)] Cluster Node [akka://System@127.0.0.1:6607]
[INFO] [03/18/2024 00:07:03.508] [System-akka.actor.internal-dispatcher-12] [Cluster(akka://System)] Cluster Node [akka://System@127.0.0.1:6607]
operation
Enter the type/key of data (temperature/humidity/pressure)
temp
Enter the operation you want to apply or 'stop': (1)
avg
Enter the window width:
5
Enter the slide:
2
Enter the operation you want to apply or 'stop': (2)
max
Enter the window width:
3
Enter the slide:
1
Enter the operation you want to apply or 'stop': (3)
stop
Enter 'operation' to create a new stream or 'exit' to terminate
```

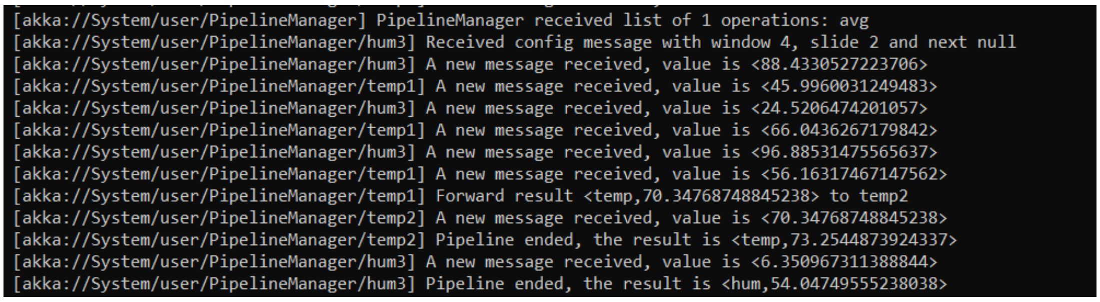
Figure 3: Commands for creating a new pipeline.



```
[akka://System/user/PipelineManager] PipelineManager received list of 2 operations: max avg
[akka://System/user/PipelineManager/temp2] Received config message with window 3, slide 2 and next null
[akka://System/user/PipelineManager/temp1] Received config message with window 5, slide 3 and next Actor[akka://System/user/PipelineMana

[akka://System/user/PipelineManager/temp1] A new message received, value is <13.935107853484885>
[akka://System/user/PipelineManager/temp1] A new message received, value is <1.823319314285754>
[akka://System/user/PipelineManager/temp1] A new message received, value is <26.845226975552283>
[akka://System/user/PipelineManager/temp1] A new message received, value is <62.28117324412155>
[akka://System/user/PipelineManager/temp1] A new message received, value is <60.81516359918125>
[akka://System/user/PipelineManager/temp1] Forward result <temp,62.28117324412155> to temp2
[akka://System/user/PipelineManager/temp2] A new message received, value is <62.28117324412155>
[akka://System/user/PipelineManager/temp1] A new message received, value is <43.48278316546143>
[akka://System/user/PipelineManager/temp1] A new message received, value is <79.12130822992137>
[akka://System/user/PipelineManager/temp1] A new message received, value is <19.115568856505092>
[akka://System/user/PipelineManager/temp1] Forward result <temp,79.12130822992137> to temp2
[akka://System/user/PipelineManager/temp2] A new message received, value is <79.12130822992137>
[akka://System/user/PipelineManager/temp1] A new message received, value is <99.34089664451375>
[akka://System/user/PipelineManager/temp1] A new message received, value is <22.912461467643453>
[akka://System/user/PipelineManager/temp1] A new message received, value is <23.345144225198577>
[akka://System/user/PipelineManager/temp1] Forward result <temp,99.34089664451375> to temp2
[akka://System/user/PipelineManager/temp2] A new message received, value is <99.34089664451375>
[akka://System/user/PipelineManager/temp2] Pipeline ended, the result is <temp,80.24779270618556>
```

Figure 4: Creating and forwarding messages between operators in the same pipeline.



```
[akka://System/user/PipelineManager] PipelineManager received list of 1 operations: avg
[akka://System/user/PipelineManager/hum3] Received config message with window 4, slide 2 and next null
[akka://System/user/PipelineManager/hum3] A new message received, value is <88.4330527223706>
[akka://System/user/PipelineManager/temp1] A new message received, value is <45.9960031249483>
[akka://System/user/PipelineManager/hum3] A new message received, value is <24.5206474201057>
[akka://System/user/PipelineManager/temp1] A new message received, value is <66.0436267179842>
[akka://System/user/PipelineManager/hum3] A new message received, value is <96.88531475565637>
[akka://System/user/PipelineManager/temp1] A new message received, value is <56.16317467147562>
[akka://System/user/PipelineManager/temp1] Forward result <temp,70.34768748845238> to temp2
[akka://System/user/PipelineManager/temp2] A new message received, value is <70.34768748845238>
[akka://System/user/PipelineManager/temp2] Pipeline ended, the result is <temp,73.2544873924337>
[akka://System/user/PipelineManager/hum3] A new message received, value is <6.350967311388844>
[akka://System/user/PipelineManager/hum3] Pipeline ended, the result is <hum,54.04749555238038>
```

Figure 5: Simultaneous handling of different pipelines composed of different instances of operators of the same type.