THEATR

An actor based language

Group members:
Beatrix Carroll (bac2108)
Suraj Keshri (skk2142)
Michael Lin (mbl2109)
Linda Ortega Cordoves (lo2258)

Goal

Create an actor-based language with fault-tolerance that simulates the logic of Erlang and the syntax of Scala in the Akka framework.

Motivation

The actor model provides a good framework for reasoning about concurrency and distributed systems. The model elegantly handles concurrency issues which can be complex when implemented in a thread-based model, such as locking and mutual access to resources. Instead of multiple threads accessing a shared resource, the actor model relies on autonomous actors performing asynchronous and independent computations.

Distributed systems requires comprehensive and secure communication capabilities with its various remote processors. To achieve this, distributed systems require that processors communicate via messages and that these messages be asynchronous. In the actor model, actors communicate with each other exclusively through messages, which are stored in mailboxes and processed asynchronously.

Description

We wish to create a programming language that implements the Actor Model with fault-tolerance.

As the Actor Model specifies, Actors will be the basic unit of computation in our language. Upon receiving a message, any actor will be able to only perform the following three actions:

1. Create another actor
2. Send message to another actor

3. Update the internal state (specify the state in which it will be when it next receives a message)

Each actor possesses an internal state. Actors will only be able to communicate with each other actors via passing messages. Each message an actor receives will be stored in its mailbox and processed asynchronously.

Programs

Our language is designed to write programs that take advantage of parallelism. Some examples of these types of programs are: chat servers, phone switches, web servers, message queues, and web crawlers. Our language is also designed for programs requiring multiple I/O computations, programs requiring strict fault tolerance, and programs requiring strict avoidance of deadlocking.

Language Details

Notable types

actor	An actor. Every actor must implement a receive() expression
int	Integer
char	ASCII character (1 byte)
Float	Floating point number
String	Strings
List[type]	Lists can be composed of multiple objects of the same type
Array[type]	Like lists, but represented in memory as a continuous block
boolean	True or false
tuple	Defined as ([type],[type]). Tuple of two objects.

Notable keywords and operators

new	To create a new object
+, -, *, /, %	Normal arithmetic operators are supported for integers

=	Assign operator
==	Check for equality operator
|	(Pipe) send message operator - message on the left, recipient on the right
=>	Used for cases in an actor's receive() clause to choose how to respond to messages. Also used to denote lambda functions
die	Kills the actor
parent	A handle to the actor that created the actor
print	Print to console
forEach	Iterates through a list or array, pass in a lambda function
main()	Entry point to the program

Common syntax:

```
int k
```

Defines an integer type called "k"

```
List[String] input
```

Defines a List of Strings called "input"

```
message findNearestNeighbor(Float datapoint)
```

Defines a message object "findNearestNeighbor" that contains a Float type called "datapoint"

```
actor Worker(int id, String name) {
	Float sum = 0.0;
	receive() {
		case findNearestNeighbor(Array[Float] input) => {
			input.forEach(f => sum += f);
		}
	}
}
```

Defines an actor named Worker that takes in an integer “id” and a String “name” to initialize it. Every actor must implement a receive() clause, which defines cases for actions upon receiving messages. Within the receive() clause, the syntax: “case [message] => {...}” defines the actions taken when a matching message is received.

```
findNearestNeighbor(datapoint) | master;
```

Sends a message of type findNearestNeighbor(datapoint) to the master actor.

Example Code:

For reference and inspiration for the actor model, we used this sample code: https://github.com/alexminnaar/ScalaML/tree/master/src/main/scala/ML4S/akka

```
// defines a message to initiate computation of nearest neighbor
message findNearestNeighbor(float datapoint)

// defines a TopK message (the result of a Worker's computation)
message resultTopK(int id, List[(String, Float)])

// defines a Worker actor
actor Worker(int id, List[Array[Float]] inputPartition, List[String] outputPartition,
int k, (Array[Float], Array[Float]) => Float distanceFn) {

        receive() {
                case findNearestNeighbor(Array[Float] input) => {
                        // finds nearest neighbor of the datapoint inside this worker's
                        // partition, sends back result as a TopK message to the master
                        print("slave ${id} received query");
                        //compute similarity of each example
                        List[(Float, String)] distances = inputPartition.map(r => distanceFn(r,
                input));
                        // Find top K classes by sorting the list by distance
                        List[(String, Float)] topKClasses = distances.enumerate.sort((idx,
                a)=>a).take(k).map((idx, a)=>(outputPartition[idx], a));
                        //send message to the parent actor
                        TopK(id, topKClasses)| parent();
                        print("slave ${id} finished nearest neighbour");
                }
        }
}

// defines a Master actor: splits the computation between Workers and aggregates results
actor Master(List[Array[Float] input,
                List[String] output,
                int k,
```

```
                (Array[Float], Array[Float]) => Float distanceFn,
                int numPartition) {

        // create partitions and workers, assign partitions to workers
        List[List[String]] outputPartitions = output.grouped(output.size /numPartitions);
        List[List[Array[Float]]] inputPartitions = input.grouped(input.size / numPartitions);
        print("data partitioned into ${inputPartitions.size} chunks");
        // create slave actors and assign a partition to each
        List[Actors] workers = inputPartitions.enumerate.forEach((idx, inputPartition) => new
Worker(randomInt(), inputPartition, outputPartitions[idx], k, distanceFn);

        print("Slave actors created");

        int slavesNotFinished = numPartitions; // keep track of number of workers finished
        List[(String, Float)]  mergedDistances; // running list of distances

        def receive = {

                // send start message to workers
                case findNearestNeighbor(float datapoint) => {
                        workers.forEach(worker => findNearestNeighbor(datapoint) | worker) }

                // receive results from workers and merge into master list to compute
                // the nearest neighbor, print out result
                case resultTopK(int id, List[(String, Float)] output) => {
                        print("slave ${id} search results received by master");
                        slavesNotFinished -= 1;
                        mergedDistances.append(output);
                        if (slavesNotFinished == 0) {
                                print("All results completed");
                                List[(String, Float)] overallTopK = mergedDistances.sort(key: (_,
                        dist)=>dist)[0:k];
                                String pred = overallTopK.groupby((a, _)=>a).map(b=>(b,
                        b.size)).sort((_,freq)=>freq)[0][0][0][0];
                                print("Prediction is: ${pred}");
                                die();
                        }
                }
        }

Float distanceFunction (Array[Float] a, Array[Float] b) {
        // definition of distance function
}

// runner code (entry point to program)
main() {
        int k = 10;
        int numPartitions = 10;
        actor master = new Master(input, inputClasses, k, distanceFunction, numPartitions);
        Array[Float] datapoint = [0.0, 2.3, 4.4]; // sample data for target to find nearest
                                                  neighbors of
        findNearestNeighbor(datapoint) | master;
}
```