NSDS AKKA PROJECT

INTRODUCTION

In this paper we present the functioning of the developed program, the design and implementation choices and how the program can work in a real environment, promoting high parallelization and computing efficiency in processing the incoming messages.

THE TECHNOLOGY

The Akka framework provides an interface to easily implement, in an high level manner, the Actor model. This model is very useful when processing messages that flow from an entity to another in a sort of stream. This is because it allows us to easily decouple the various phases of elaborating a certain data, by sending messages that has the current state of the data, and not sharing the internal state of each actor that communicate with others, assuring that every actor is autonomous and the operations happen asynchronously.

SYSTEM DESIGN

The system runs on a hierarchy, or tree, of actors (Figure 1.1), each of which has it's own task, and running independently from the others. At the top of the hierarchy there's the StreamManager actor, that controls the startup of the system. It is the landing point of the communication, since its main task is to send to the right partition each incoming message. We define a partition as a cluster of actors, managed by a TopicManager, that receives and elaborates only a certain type of message, identified by a topic that belongs to the set of topics T={"Temperature", "Humidity", "Pressure"}. The message propagates to the leaves of the formed tree, which are the operators that actually transform and aggregate the data in order to forward that to the FileWriterActor that eventually write the aggregated data to a json file to have a persistent trace of the elaboration done. Moreover the TopicManager distributes the incoming messages using a Round Robin strategy in order to provide a load balancing between the different actors of each topic. To ensure that the system is fault tolerant each actor has its own supervisor that is child to the supervisor of the above level. For example we have the OperatorSupervisor actor that supervise the operators, which is the child of the TopicSupervisor, the supervisor of the topic managers. This is done both to guarantee that the actors are actually fault tolerant, using a restart strategy each time a fault happens, and also to guarantee that the supervisors itself is fault tolerant to some extent.
It is worth to say that the FileWriterActor is detached from the other actors since it does not properly belong to the system stream that elaborates the data, but is just an entity that take the refined data and persist it.

DATA STRUCTURES

During the execution of the program various types of messages are used, such as to ensure that a certain actor is correctly initialized or to start sub actors of a determined

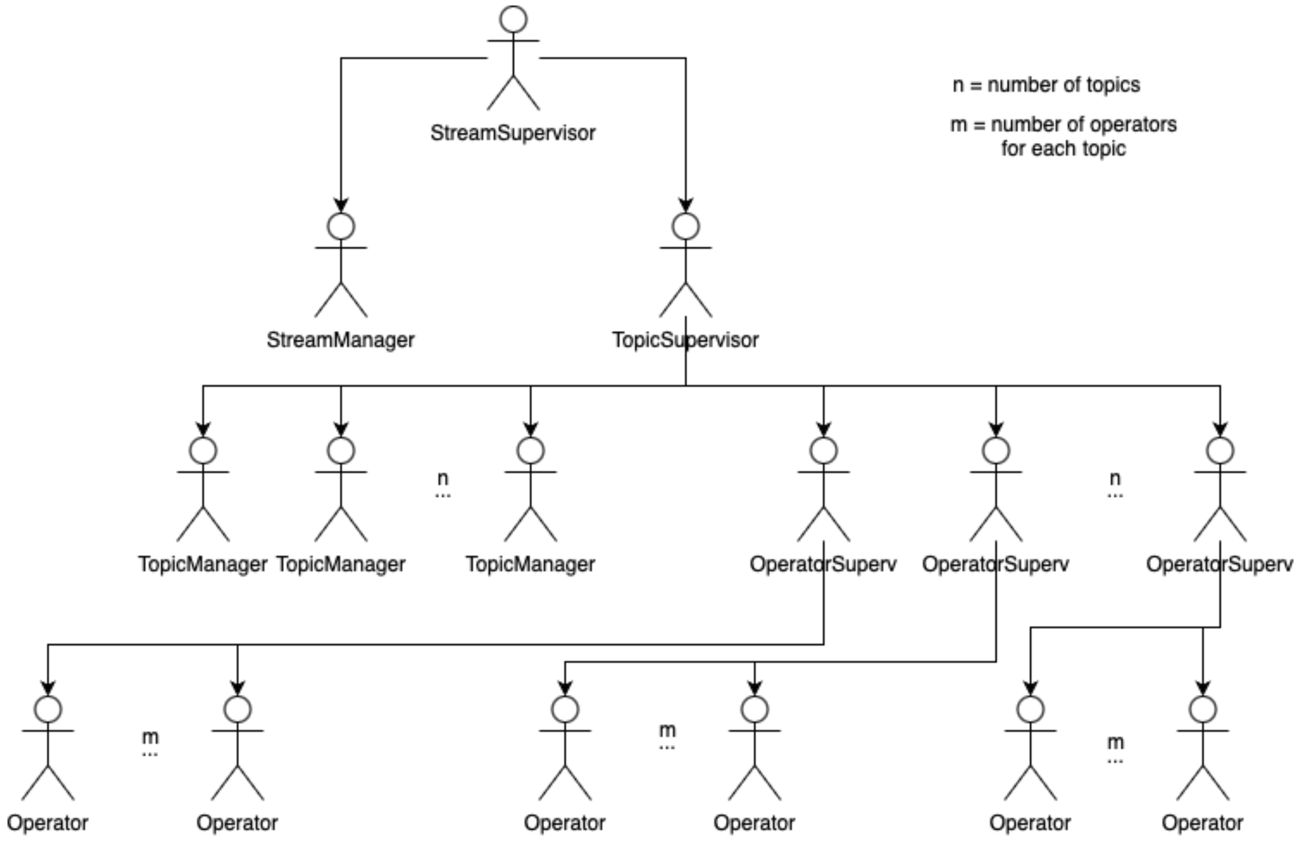

Figure 1.1

actor, but the main messages that contain the effective data to be elaborated are two. SensorData and Aggregated data, which represent respectively the data obtained from an external sensor and the data aggregated to form significant informations.

The transmitted messages are objects that have mainly three attributes, and represent measurements done by a certain sensor. They are identified by:

- The topic to which they belong, identified by a key
- The actual measurement value
- A sequence used as a unique identifier

The aggregated data are instead identified by:

- The topic to which they belong, identified by a key
- The value of the aggregation
- A sequence number used as a unique identifier
- The timestamp in which they have been produced
- The type of aggregation performed

In order to send the SensorData from a client (sensor) to the system, we have to serialize them into json strings, that are then deserialized by the system into objects in order to use them.

The aggregation is done through a sliding window, which is modeled using an object that stores the values up to the window size and, when the window is full, performs the slide of the values that have to be aggregated. The window is characterized by:

- The max size of the window
- The slide that has to be performed, expressed as an integer
- The elements present in the window itself, stored in an array
- A counter to keep track of the number of elements present into the window

SYSTEM FUNCTIONING

During the system bootstrap, the program reads the configuration file to know about the hierarchy structure (e.g. number of operators for each partition, number of partitions, window size for each partition, host, port, etc.). Then the top supervisor of the stream is initialized and all the others are initialized in chain. Also the FileWriter is initialized as a separate actor under the user guardian actor.
Each top actor has a local representation of the actors that are under him, since they are not directly its children, which could be a Map, for the topic managers since each topic has to be mapped to its manager, or a simple list, such as the one that yields all the operators under a certain partition.
When the message arrives the system has the Stream manager as a landing point. The actor job is to send the messages to the right partition of the system, that is identified by the topic, or key, of the message. Once the message is sent to the right partition the corresponding Topic manager will send it to the operators, following a Round Robin strategy to guarantee that the messages are evenly spread among all the operators of the partition. The operators gather the incoming messages and put them into their local window, each operating independently from the other.
Once the window of an operator is full, the operator performs the aggregation function associated to the specific partition, by sliding the window and aggregating the slides values. It encapsulates the result in an AggregatedData object in order to forward it to the Topic Manager to finally send it to the FileWriter actor, whose task is to update, or eventually create, the json file that stores the results of the aggregations.
In case of crush of any of the actors, the system supervisors proceed to restart the faulty actor, resuming the state in which it was with its mailbox. For ease of functioning the system is shutdown by the Stream manager once the FileWriter doesn’t receive a message to write for more than 20 seconds. However in real scenarios the system must be continuously up since we don’t know when a message is coming, and the system cannot be stopped after just 20 seconds.

Running example

The client starts sending messages to the system. Let’s focus on a single message. The topic to which it belongs is, for example, the “temperature” topic. The serialized message arrives, the system deserialize it, and it’s immediately caught by the Stream manager. Then the actor looks at the key of the message and redirects it to the corresponding partition, in this case the “temperature” one. The message is sent from the Stream manager to the correct temperature topic manager, that computes to which underlying operator forward the message thanks to the round robin policy, and forward the message. Once it is received by the operator, the message is analyzed and its value is inserted in the window representation of the actor. If the inserted value fills the window, the inner values are aggregated and then the window is slided, else just the internal state of the window is updated. Once the data are aggregated the are put inside an AggregatedData object, along with a timestamp and an identification number, which is then sent to the sender actor, the “temperature” manager in this case, the will forward it to the final actor of this pipeline, the FileWriterActor which will write the aggregated data into a json file.

AKKA DESIGN CHOICES

Akka is usually used for the performance that it provides since the actors operates asynchronously over different threads in a non-deterministic fashion. In our scenario we identify a few critical steps that have to be performed in a deterministic way, implementing a sort of serialization of the steps since we don’t want that our system incur into racing conditions, where some actor that has not finished its initialization is asked to perform some kind of computation. This issue arise mainly when initializing the hierarchy of the system since we have some exchanges of messages between actors that might be not initialized yet. To solve this issue we use the Ask pattern provided by akka. This pattern allows to freeze the pending operations of the actor system context in which we are and wait for some action to be resolved, limiting concurrency issues between actors. However we have to be careful with this pattern since it serializes something that is meant to be executed in parallel, slowing down the system. Hence this pattern is only used at system startup to guarantee the correct initialization of the actor hierarchy.