Compute Infrastructure

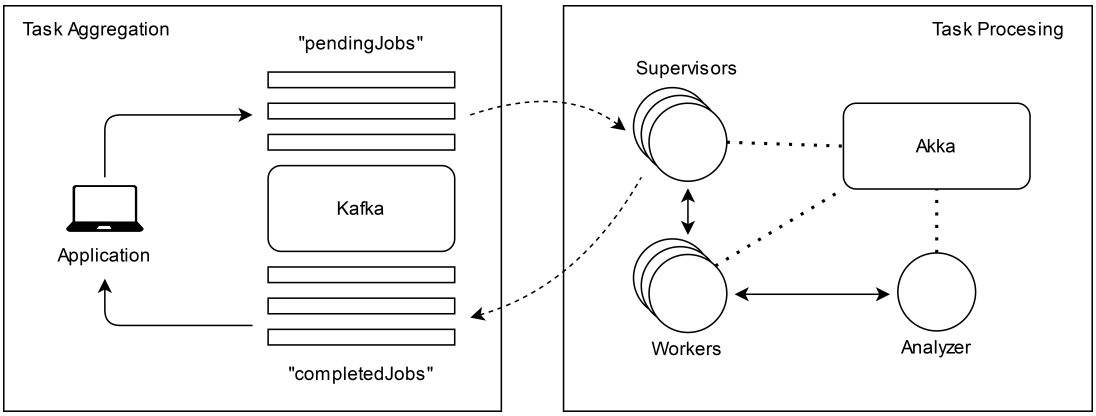
Introduction

This project aims to design and implement a system that accepts compute tasks from clients and executes them on a pool of processes. Each request from a client includes the name of the task to be executed (e.g. image compression), a payload with the parameters to be used in the computation (e.g. a set of images and a compression ratio), and the name of a directory where to store results.

Architecture

The distributed system is split into two modules:

- A **task aggregation module** based on Apache Kafka.
- A **task performance module** based on Akka.



The communication between modules is performed through the Internet.

The system is designed to be non-blocking for the final user. The queue between the front-end and the back-end allows the user to request the execution of a task and keep using the front-end application while the task is being performed.

Task aggregation module (Kafka)

This module consists of a Kafka cluster with two topics:

- `pendingJobs` acts as a queue storing task performance requests from the users.
- `completedJobs` is a list of result folders where users can find the outcome of their requests.

Users can access Kafka topics through an application that serializes their requests to be correctly handled by the back-end module. This same application monitors the completed tasks to notify the user when its request is fulfilled.

When a new request by a user is generated, it is associated with a key that identifies that specific task. When an event with that key is published on the list of completed tasks, the application can locate it for the user and show a notification.

Task performance module (Akka)

This module consists of an Akka cluster with a predefined number of simultaneous working blocks and an analyzer actor that computes some metrics on the task requested by the users. By "working block" we mean a pair of actors (a supervisor and a worker) that can process a single task.

- The supervisor interfaces with Kafka and acts as a consumer for the topic `pendingJobs`. Once it polls an event it checks if the payload is well-formed and possibly forwards it to its worker. When the worker finishes the task the supervisor gets notified and publishes a completion event on the topic `completedJobs`.
- The worker mimics the execution of the received task and notifies his supervisor once it finishes the computation.

Supervisors and workers are two distinct entities because the latter could eventually fail and need to be restarted. Without a supervisor, it'd be much more difficult to realize that a worker has crashed. When a worker crashes, the supervisor takes care of handling the incomplete tasks sending them again to the worker actor once it is restarted.

Design choices

Kafka for task aggregation

The main factor that led to this choice is the storage capability of Kafka which makes it perfect for the implementation of a queue. The use of Kafka allowed load balancing through the feature of consumer groups and made the whole system intrinsically asynchronous. Moreover, Kafka has a very powerful horizontal scalability so when the system grows in size it's enough to distribute the Kafka cluster and replicate the nodes to hold the load.

Akka for task performance

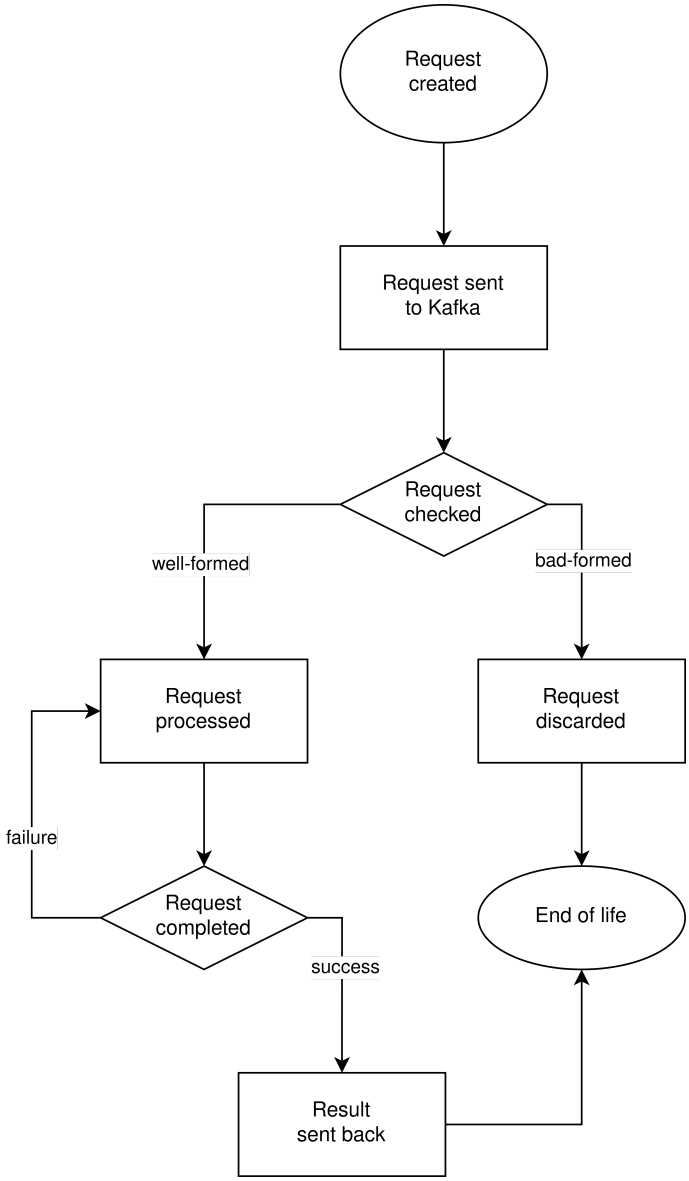
Since the specification requires scheduling the task execution onto a set of processes, the choice of Akka was the most fitting. The actor paradigm of Kafka is symmetric with the need to mimic multiple available processes for task performance. This choice also allowed the implementation of the strategies for failure-handling since Akka provides the needed primitives for this purpose.

Why not Spark?

Even if Spark is very powerful when it comes to task performance, the scenario for this application didn't fit the Spark typical use-case. Spark is very good at processing pre-deployed operations on a huge amount of data, while in this application the amount of data isn't a problem since the information exchanged between the user and the back-end are only task performance requests. Moreover, when a failure occurs in Spark, it is necessary to reboot the whole system. In the actual scenario of this application, failures may affect only a partition of the system.

Main functionalities

It is meaningful, for this application, to show the flowchart of the life of each user request. The overall picture of the system is given by the union of many requests processed simultaneously.



Conclusions

The intrinsic modularity of the systems allows deployment without bothering with the links between modules. Each module can be run separately from the others since the system is fully asynchronous. Each module was independently tested with corner cases and in a maximum stress environment.

Authors: Simone Braga, Alessandro Bertulli, Marco Dottor