Middleware Technologies for Distributed Systems

Project #3 - Compute Infrastructure

Additional feature: Logging and analysis

Samuele Negrini, Samuele Pasini, Giorgio Piazza

1 Introduction

Compute infrastructure is a system that provides computing powers to users. They can submit to it a task among a specified set of tasks. Each submitted task will be identified inside the system with a Universally Unique Identifier (UUID).

The system has a pool of available processes and each submitted task will be executed onto a single process taken from the pool. If the number of submitted tasks exceeds the number of available processes, exceeding tasks will be put into a waiting queue with FIFO policy and priority set to *normal*.

In case a process fails during the execution, it is resubmitted into the waiting queue in a silent way (i.e. no notification of the failure is sent to the user) and its priority is increased to *high*. Tasks can be submitted to the system using a web interface.

Each request from a client includes the name of the task to be executed, a payload with the parameters to be used in the computation and the name of a directory where to store results.
Users will receive a notification when the task they submitted has completed the execution and the result has been successfully stored to the disk.

The system logs information about task life-cycle, analyzing periodically statistics and storing results in a database.

2 Architecture

2.1 Assumptions

Before showing the overall architecture, it is necessary to state some assumptions. Please see Section 4 to see future improvements and limitations.

1. Cluster nodes cannot fail during task execution (while processes can fail at any time).
2. Inputs submitted by the users are valid.
3. Time to compute a task $\gg$ time to connect to task processing status page (tasks are heavy).

2.2 Overall architecture

The system architecture is divided into two platforms (Fig. 1):

- **Computing platform**: contains all the logic to handle requests from clients, executes them on a pool of processes and stores the results on disk.
- **Logging platform**: contains all the logic to perform operations and analysis on the system's logs.

2.2.1 Computing platform

This platform is based on the actor model implemented with a cluster architecture. A cluster is a dynamic group of nodes. On each node there is an actor system that listens on the network and manages one or more actors.
The cluster is composed of three type of nodes (Fig. 2):

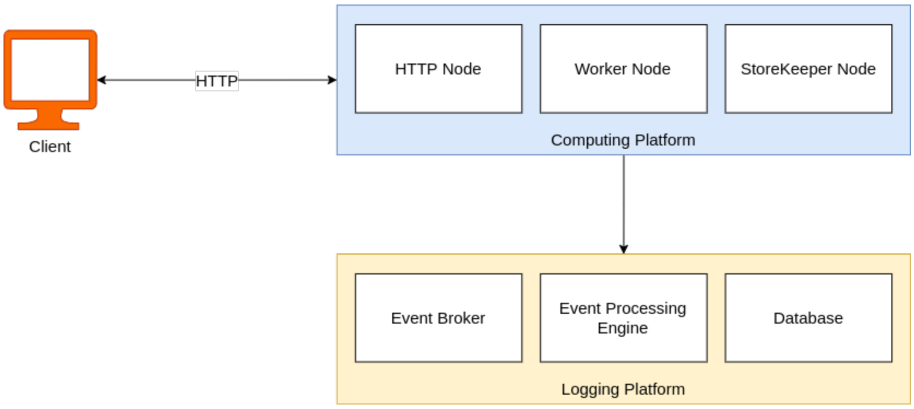

Figure 1: System architecture

- **HTTP Node**: handles the HTTP requests from the client and forwards them to the computational nodes (Worker Node) by using the dedicated WorkerRouter. The WorkerRouter is a particular running actor that will forward any message sent to it to one final recipient out of the set of routees. In particular it adopts the round robin group strategy to forward messages to the BalancingPoolRouter inside the Worker Node.
  Moreover, it manages a Server-Sent Events (SSE) with the client so that it can notify the user task completion.
  Http actor logs the UUID of the accepted tasks on the *pending* Kafka topic.
- **Worker Node**: receives the tasks and puts them in a custom mailbox of the BalancingPoolRouter. Then the router will fetch the first task from the queue and submit to one of the available worker actors (processes). Once a worker finishes a task it sends the result to the StoreKeeper Node.
  Worker actors log the UUID of the started tasks on the *starting* Kafka topic.
- **StoreKeeper Node**: receives the result of the computation from a worker and stores it on his local storage. Then it contacts the HTTP Node through a dedicated router and sends a notification of task completion.
  StoreKeeper actor logs the UUID of the completed tasks on the *completed* Kafka topic.

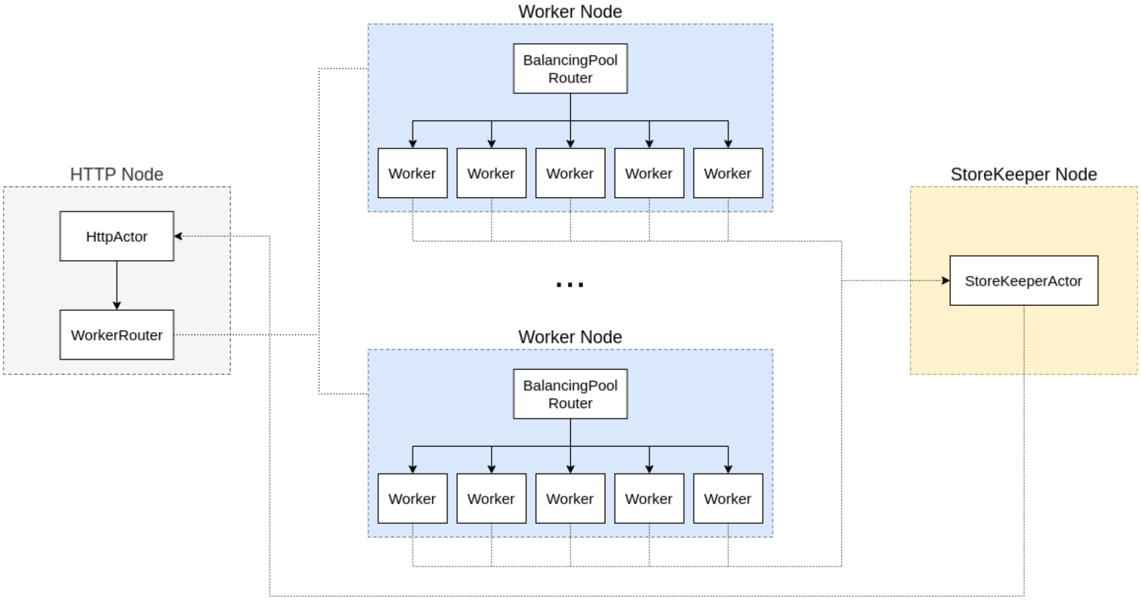

Figure 2: Akka Cluster

2.2.2 Logging platform

This platform follows an **Event-Driven architecture**. There are 3 types of events (produced by the computing platform) and corresponding topics:

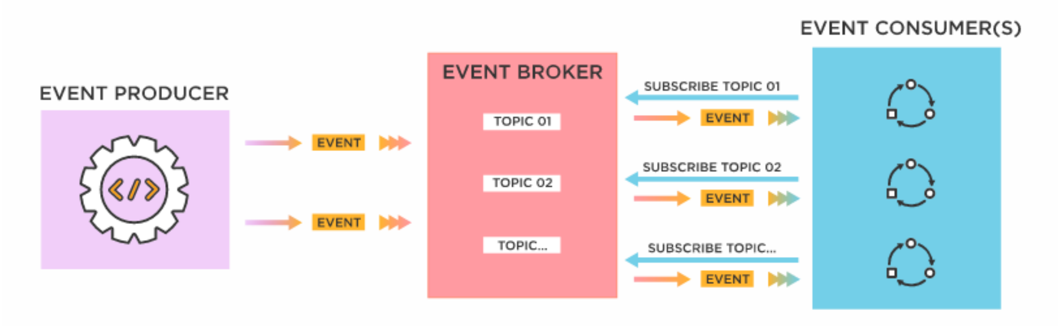

Figure 3: Event-driven architecture

- A task is received by the computing platform
- The execution of a task is started
- The execution of a task is completed (it is considered completed when the result is stored)

Every single event in the topic is characterized by its ID.
Every process of the computing platform is considered to be a producer.
The consumer is a processing engine that subscribes to the topics and does micro-batch processing executing repeatedly the required queries over the topics, treated as streams of events. To process the streams there is a **cluster** (see Figure 4) composed by a master and slaves. The driver program is submitted to the cluster that schedules processing tasks on available slaves.
Since in our scenario we do not have to compute very large datasets and we are constrained by the power of our machines, we consider just a single Worker.

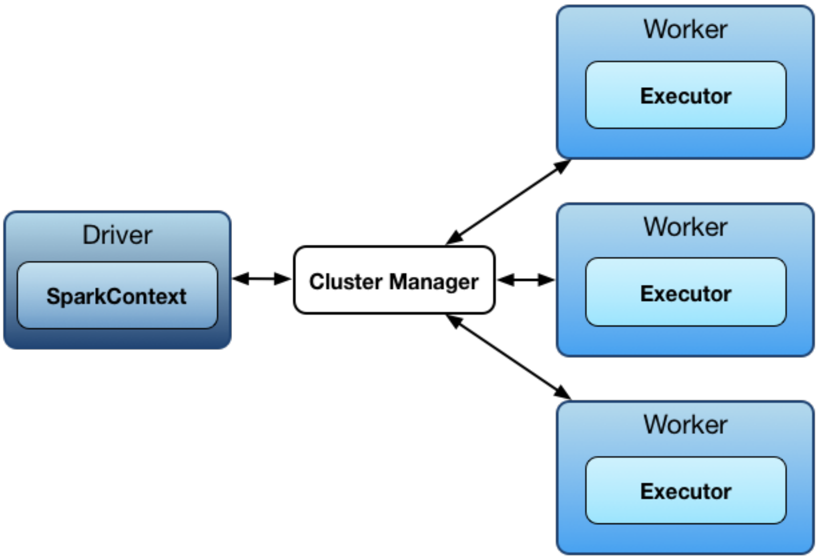

Figure 4: Processing Stream Cluster

It is also needed a **sink** to write the output of the queries. The cluster connects to a **database** and writes the output of the queries in the related tables.
To implement the logging platform we used Apache Kafka as Event Broker and Apache Spark as Event Processing Engine. See 3 for more details.

2.3 Interfaces

2.3.1 HTTP Interface - Akka

The HTTP interface is provided by AkkaHTTP which starts a web server and forwards the HTTP requests to an actor in order to process them. The requests are provided to the HttpActor via *Ask Pattern* in order to wait the response to be sent to the user.

2.3.2 Akka Actor - Kafka

Akka actors could publish events in the relative topic using the Producer API offered by the Kafka. Given the Kafka Producer API, there is the possibility to insert Key-Value pairs. In the application, events are published only with the UUID of the task as the value. Other useful fields, like the timestamp of the publication, are automatically inserted in the published event.

2.3.3 Kafka - Spark

Spark could be integrated with Kafka thanks to Structured Streaming API. With this approach a Kafka topic is represented as source of Stream and the events are read as Structured Streaming.

2.3.4 Spark - Database

Spark uses a MySQL driver (JDBC) to open connections to database in order to use it as sink.

3 Design choices

3.1 Akka

For the computing platform we have decided to adopt Akka. Akka implements the actor model that allows to manage concurrent computation in an easy way.
Moreover with Akka we get scalability and fault-tolerance features that satisfy our requirements.
Furthermore Akka provides a module (Akka HTTP) to instantiate a web server and to be able to receive requests via HTTP and have them processed by an actor.

3.1.1 Akka vs MPI

Another way to distribute the computation could be to implement an application based on MPI. We discarded this option as MPI does not guarantee an out-of-the-box fault-tolerance system.

3.1.2 Akka vs Kafka + Microservices

This system could also be implemented by using a microservice architecture. In this architecture is possible to decouple the microservices using Kafka. This is for sure a good alternative to our Akka approach. However, in our analysis the microservice architecture is extremely good when there are multiple services that should perform different tasks while keeping communication with each others.
This will fit in an architecure where a microservice A is specialized in tasks of type A, while in our system we have a pool of **interchangeable** processes that can execute a task taken from a finite set of tasks (potentially of type A, B, C, ...).

3.1.3 Akka clustering

We decided to use Akka clustering to ensure the possibility of expanding the computational capacity of the system as needed.
A cluster makes it possible to dynamically grow and shrink the number of nodes used by a distributed application, and removes the fear of a single point of failure. Indeed, if necessary, it is possible to spawn a new Worker Node that increases the number of processes available within the system. Akka natively provides methods for creating a cluster of nodes and manages communication between them through the use of actors and routers.

3.2 Apache Kafka

The goal of the logging platform is to write sequences of events that will be processed and analyzed according to the event-driven architecture (Fig. 3) described before.

A traditional approach could consist in collecting physical log files from the servers and putting them in a distributed file system (like HDFS from Apache Hadoop) for processing.

Our approach adopts Apache Kafka that allows us to abstract the level of detail of files and manage the logs as **stream of events**. Furthermore, Apache Kafka can be easily integrated with some processing tool (e.g. **Apache Spark**) to process the streams of events.
This is made simple with the **publish-subscribe** pattern used on topics:

- Events are published on a topic by Akka Actors (Producer)
- Events are consumed from a topic by Spark (Consumer)

Thanks to the Apache Kafka architecture, we can also benefit from several advantages like fault tolerance, low latency, scalability, durability and persistency.

3.3 Apache Spark

Given the logs written in Kafka topics we need a processing engine to analyze them.

Apache Spark is an extension of the **Map-Reduce** approach used to tackle diffuse computation in Big-Data environment. Apache Spark allows to distribute computation over a cluster of workers with support for caching, scalability and fault-tolerance.

In particular we adopted **Spark Structured Streaming**, a scalable and fault-tolerant stream processing engine built upon Spark SQL Engine.
With this approach the input streams (i.e. the stream of events in the Kafka topics) are treated as unbounded tables where new events are appended.
From a physical point of view, the entire table is not materialized. Spark reads only the latest available data from the source, updates the result incrementally and discards the data: only the minimal intermediate state data required to update the result is maintained. In the application we registered several queries that will be executed multiple times based on Trigger time.

3.3.1 Spark vs MPI

It seems possible to implement a similar behavior with MPI, but generally it has a different purpose. With MPI we have explicit parallelism, communication and synchronization, using low-level primitives and maximizing the usage of resources without fault-tolerance. This is useful when the focus is on the performance, like in population dynamics. Instead, we are more interested in an easier use, with high-level primitives, dynamic tasks and fault-tolerance. This is the reason why we choosed Spark over MPI.

3.3.2 Database Sink

The database used to write the output of the queries is **MySQL**. To run arbitrary computation as output sink it is needed to use a *foreachBatch* sink. It is important to notice that the queries are independent of the Sink, so an interface exposes the sink methods following the **Strategy Pattern** and the implementation manages the write operations in MySQL database.

Since the connection to the database could become a bottleneck for the tasks, it is required an efficient usage of the resources. **HikariCP** is used as connection pool and, as suggested by the documentation, the connection is acquired and released to the pool using *foreachPartition* method.

4 Conclusion

The project aims to show the usage of the technologies explained during the course and their integration in an architecture able to satisfy the requirements, not to build a product ready-for-the-market. In particular, the implementation is able to:

- Handle simultaneous tasks submission
- Distribute task execution over a cluster
- Handle a centralized storage of the computed result
- Handle process failures and restarts
- Log data about tasks
- Analyze logs and extract required statistics
- Store the result of queries on Log events in a database

4.1 Future improvements

The presented architecture could be improved in different ways.

- Create a system to retrieve the results of the computation stored on disk
- Create a distributed file system to store results
- Handle failures at node level
- Balance the computation considering the effective usage of resources in different nodes