DOI 10.5162/ettc2020/6.7

Job Process Architecture Based On PostgreSQL Database

Gonzalez-Martin Moises
Roses-Sanchez Diego
Garcia-Garcia Jorge (ALTRAN)
Flight Test – Airbus Defence and Space
moises.gonzalez@airbus.com
diego.roses@airbus.com
jorge.garciagarcia@altran.com

Abstract: The purpose of this paper is to show an integrated Job Process architecture that allows executing heavy jobs that can be parallelized at process level without the need to use complete Big Data Architecture. This solution can be seamlessly integrated into any Data Centre, be it in a Linux or Windows environment. PostgreSQL is shown as the system to coordinate the workflow of execution of these processes.

Key words: Job, Process, PostgreSQL[1], Parallelization, Batch processing

Introduction

The flight test world has to deal with large amounts of information coming from test activity recordings, which needs to be analysed in different ways. Many of the analysis involve heavy computational processes that take long periods of time, and are repeated many times with different sets of data. When there are a high number of test activities that need to be processed, it can take many days to get the results.

There are many Big Data and cloud computing platforms like Hadoop YARN[2] or Microsoft Azure Batch[3] that could handle these processes, but there could be some downsides: complex, expensive and cloud dependent that may not be allowed when it comes to military data.

The Job Process architecture proposed enables the parallelization of data analysis processes on a virtualized computing platform to accelerate results and report generation.

Job Process Architecture

To provide a simple solution, we propose to create a task queue in a database that will run on a set of virtual machines connected to the "local" network.

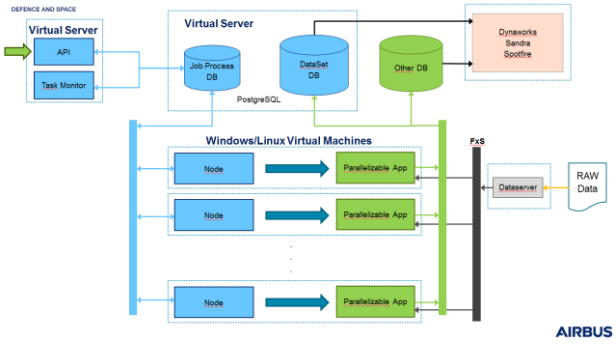

Figure 1. Job Process Architecture

Here, we can identify some definitions:

- **JOB**: A group of tasks that could have a specific order of execution.
- **TASK**: The minimal piece of analysis, performed by a console executable process.
- **NODE**: A Windows/Linux computer connected to the same network, running a "Job Process" Windows Service or Linux Daemon for task queue execution.
- **Parallelizable App**: Any app accessible from the "local" network such as Matlab or Python processes that can be run in console mode, receives an argument in JSON[4] for the inputs, and generates results unattended.

- **FxS Dataserver**: Provides access to test activity recordings.
- **DataSet DB**: The system provides a database to store task results.

The core of this architecture is the service/daemon running on the Nodes. It will periodically query the database for available task that comply with a set of rules. Once a task is found, it will be locked down to avoid double execution and if lock is granted, the executable process associated to the task will be run in the node. Once the task is completed, its status is updated on the database and the service will continue looking for new tasks.

Database structure

Figure 2 shows the tables used to handle the Jobs scheduling and processing.

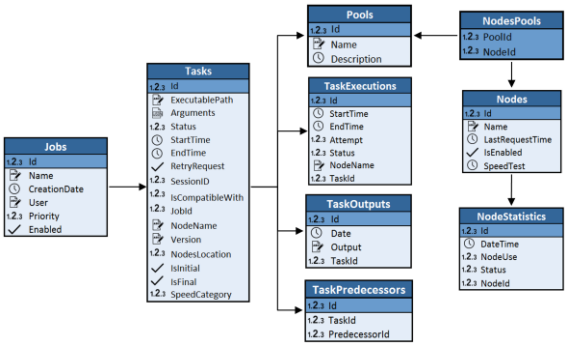

Figure 2 Job Process Database diagram

JOBS

Defines groups of tasks requested by a User with a specific priority level.

TASKS

Each of the processes that comprise the Job, with information to run the process and properties that deterime when and where this task will be executed.

POOLS

It allows the reservation of a set of computers for the execution of specific tasks, generally for any type of analysis that should have priority over any other task.

TASKEXECUTIONS

Used as a locking mechanism to prevent multiple nodes executing the same task. An index on TaskId, Status and Attempt fields, ensures that only one Node will be able to lock the task for execution.

TASKOUTPUTS

Stores the standard output of the task process if any. It could be useful to trace execution errors.

TASKPREDECESSORS

Defines the order of execution of tasks for a given Job.

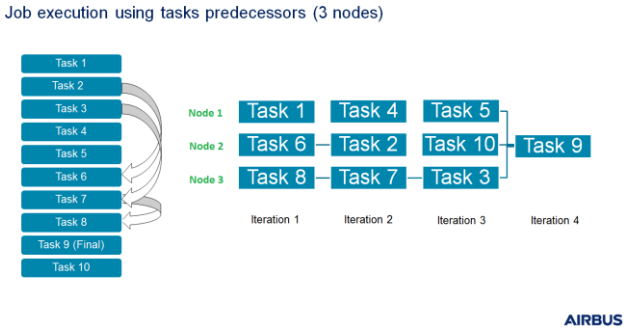

Figure 3. Task Precedence example

Figure 3 illustrates an example of the order of task execution on 3 nodes with predecessor tasks defined. This is what happens in the first iteration:

Task1 has no predecessors so it will be the first executed task by node 1.

Node 2 will try to execute Task 2, but it needs that Task 6 be executed first so, it will run Task 6.

Node 3 will check Task 3, which needs Task 7 to be executed first, and it also needs Task 8 to be executed first, so it will run Task 8.

The same process continues in the next iterations, if the Task has a predecessor, it will try to run that task instead. If a task is set as Final, it won't run until all other tasks are completed. Also, if there is an Initial task, no task will be ran until that task execution has finished.

NODES

The Windows service / Daemon installed on the Nodes stores information about the characteristics and status of the Host in this table. It is used by the Job Monitor Service to detect blocked tasks or nodes.

NODESTATISTICS

Used by Job Monitor Service to store information about task execution performance to generate statistical reports.

API

The System exposes an API that allows users to develop tools to manage and request Job execution.

It also allows gathering information about the system and the status of jobs, tasks and nodes.

JOBPROCESS Service / Daemon

This is the core of the architecture. It is a service that can be installed on any computer connected to the local network. It turns it into a Node that will start looking for tasks to run, making system scalability really simple.

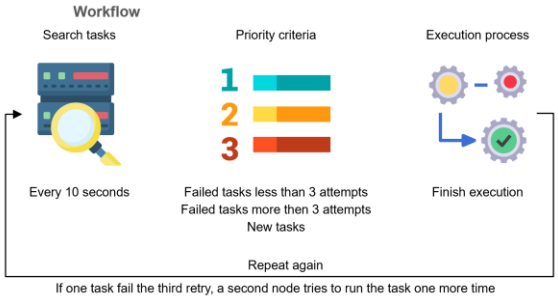

Figure 4 Service/Daemon workflow

The service queries the database every 10 seconds looking for a task suitable to execute.

FAIL CHECKS

Since the tasks that make up the jobs are external processes to the system architecture, we must bear in mind that these tasks may not work correctly, and some node may be blocked for uncontrolled reasons, so the system must be prepared to handle these situations.

Every task has 3 attempts to complete on the node that has picked it. If it continues to fail, a different node will try to run it once. This is to discard the issue is related to the node. The TaskOutputs table contains information about the task execution and could help to investigate issues.

When using the system DataSet database to store the results, if a task fails to execute, the Job Monitor service reverts the results to clear any incomplete data.

JOB MONITOR Service

This service runs in the computer where the API is hosted and it is responsible of system health monitoring.

Its main functions are:

1. Gather Node task execution statistics every minute.
2. Monitor and reset tasks running on nodes that has been unresponsive for more than 40 seconds, and rollback partial results.
3. Restart blocked nodes (available for VMWare virtual machines).

Conclusion

The Job Process System Architecture can be easily implemented to allow scheduling heavy computational tasks such as flight data analysis. Also, it can be used to run any kind of task that involves an unattended console App with JSON inputs.

Nodes can be created either with physical or virtual devices increasing the parallelization capacity to reduce Job execution times. Installation of the JobProcess service/daemon turns any computer connected to the system network into a node that will execute tasks.

Providing “intelligence” to nodes instead of having a task dispatcher makes the system simpler and easier to scale, and having a distributed architecture, a critical point of failure is avoided.

References

[1] https://www.postgresql.org/

[2] https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html

[3] http://azure.microsoft.com/en-gb/services/batch/

[4] https://www.json.org/json-en.html