Evaluating Big Data Frameworks To Simplify Distributed Task Execution In Apache Airavata

Apoorv Palkar
*Department of Computer Science
University of Illinois at Urbana-Champaign
Champaign, IL 61820
apoorvp2@illinois.edu

Gourav Shenoy
†Science Gateways Research Center
Pervasive Technology Institute
Indiana University
Bloomington, IN 47408
goshenoy@indiana.edu

Ajinkya Dhamnaskar
†Science Gateways Research Center
Pervasive Technology Institute
Indiana University
Bloomington, IN 47408
adhamnas@iu.edu

Suresh Marru
†Science Gateways Research Center
Pervasive Technology Institute
Indiana University
Bloomington, IN 47408
smarru@iu.edu

Marlon Pierce
†Science Gateways Research Center
Pervasive Technology Institute
Indiana University
Bloomington, IN 47408
marpierc@iu.edu

***Abstract*—Apache Airavata powers users to run scientific applications on remote computing resources. Airavata integrates with science gateways to moderate computational jobs on diverse resources. Some of the job may require to execute multiple tasks. Newer technology trends such as Cloud Computing, BigData and Internet of Things are yielding open source software frameworks which are built for scale and reliability. In this paper we share our experiences in exploring the use of such frameworks and adapt for science gateway task management needs implemented by Apache Airavata.**

I. Introduction

Apache Airavata is middleware that supports science gateways by implementing common features such as job submission and metadata management [1], [2]. Airavata has several internal components, including a Registry, an Orchestrator, and a Task Execution engine, that are accessed through an API. These components may be implemented using a number of strategies and communication patterns. The internal components, for example, may be implemented as containerized microservices that communicate through messaging system, through direct remote procedure calls, or through REST calls [3], [2]. Components can be replicated for fault tolerance and load balancing.

Internal operations for even basic Airavata interactions may be thought of a Directed Acyclic Graph (DAG). We note here that the DAG may correspond to an scientific workflow (such as staging data in and out of resources and executing operations on the remote resources), but we are concerned here with the internal management of the DAG and how it involves multiple Airavata components.

Presented at Gateways 2017, University of Michigan, Ann Arbor, MI, October 23-25, 2017. https://gateways2017.figshare.com/

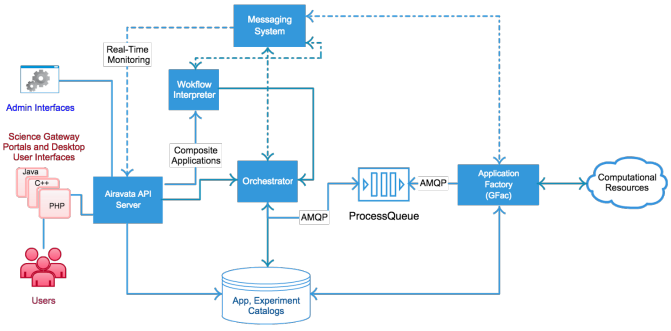

Fig. 1. Apache Airavata conceptual framework.

The challenge for Airavata is to simplify its internal communications. Can we identify a framework that will manage the lifecycle of Airavata components and the flow of information between them without having to implement this ourselves? That is, we would like to identify a framework that can provide fault tolerance, load balancing, and elasticity of components like the Task Executor: if a Task Executor instance fails, start a new one and route work elsewhere. The desired framework should also preserve state in the DAG of a particular request and handle errors. We may term such systems as 'fabric'or 'substrate'frameworks.

The current implementation uses a custom distributed task manager [4]. This, however, presents a significant drawback. The implementation code required for this section of Airavata is large. It requires a great deal of maintenance. In addition to this, it presents a learning curve for future Airavata developers.We want to keep the distributed task manager as simple as possible.

There exist many general purpose distributed schedulers

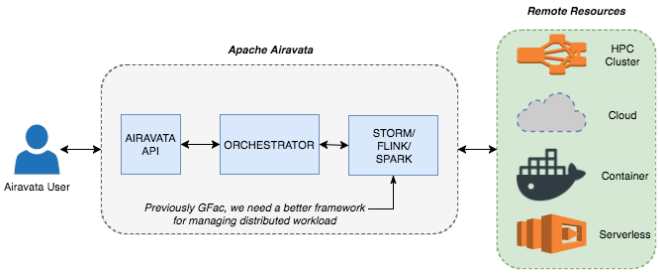

Fig. 2. Where the current work fits in Airavata abstract architecture

[5]; however, none fit our use case perfectly. Such schedulers include Apache Mesos [6], Yarn, and Spark standalone. Our use case demands a much more higher level overview. These schedulers conduct scheduling on a far lower level. As a result, we have explored other sister frameworks that may fit our need. Apache Storm [7], Apache Spark [8], and Apache Flink [9] are examples of popular open source technologies that solve problems similar to the ones we have outlined above These three technologies focus primarily on event processing and batch processing. We are not, however, interested in their common use cases. All three technologies are able to take a modified input and then perform black box operations and execute actions on a cluster. Instead of executing on a cluster, we want one of these frameworks to take a particular DAG and manage its internal state and orchestrate its passing to the task executors for final execution.

This paper examines each of these technologies as a possible substrate for Apache Airavata components. There is significant overlap in the high level goals of all three:

- **Fault tolerance:** It is of the highest essence that there is fault tolerance in a framework. There will potentially be situations where tasks executors fail or face issues regarding server/HPC connections. The most effective frameworks should handle these cases internally without external development.
- **Ease of Use:** The entire point of this effort is to improve functionality, decrease development effort, and decrease the amount of extraneous code. A successful framework that replaces the current implementation should be easy to deploy and manage in the long term.
- **Scalable:** It should be easy to increase the length and amount of tasks and jobs. The increased overhead should not be an issue.

These three qualities are highly desirable for our use case. Simplifying the current code and replacing it with one of these frameworks would promote efficient development, high reliability, and a small learning curve.

II. Apache Storm

Apache Storm is a distributed stream processing framework. It was designed to handle real time data and perform fast data analytics. Storm performs many of the desired operations of our required use case. This includes high scalability, ease of use, robust fault-tolerance, and very thorough documentation. Storms signature development is through its use of *bolts* and *spouts*. A spout is a source of information input. This can include information such as Twitter tweets, Facebook posts, MongoDB, or a simple .txt file. Storm has the ability to deal with this information efficiently and process it through internal channels.

Our concern is not actually how Storm achieves this high availability and efficiency. We are interested in our use case with Storm. We may think of the flow of information between Airavata components associated with a task request as a stream. Internally, a Storm application uses *topologies* in the form of a DAG combined with spouts and bolts. A DAG is called a *topology* in a Storm context. The topology is structured to flow similar to a data pipeline. Similar to a MapReduce job, Storm performs similar steps to perform calculations. The main difference between MapReduce and Storm is that Storm performs real time calculations instead of using small batch processing similar to Hadoop/MapReduce.

We can potentially use Storm to write spouts to take input data and pass it to bolts which handle the execution. A bolt is executed via the *.setBolt(*)* method. Potentially, we can devise our custom topology from the Airavata use case and pass it to Storm. A series of *setBolt* statements will execute and handle the execution. This is easy to implement.

III. Apache Spark

Spark is similar to Storm but has a much broader implication in its use case. Spark is a general purpose cluster execution framework. It provides many libraries such as Spark Streaming, GraphX, and MLib. We want to analyze Spark to see if it can process a DAG and execute the series of tasks and jobs given to it as input. One can start a Spark context and execute Java statements to control the flow of input.

Spark provides excellent support particularly for Cassandra. Cassandra can be used to store and retrieve Airavatas internal DAGs. Spark provides innate functions such as map(), filter(), reduceByKey(), and cache(), which allow us to isolate each segment or task in one job and then execute it. In addition to this, Spark also meets all our criteria for an effective framework similar to Storm. The documentation is also one of the best among Apache projects.

The ease of use is a relative drawback however. Because Spark is such general purpose framework, it provides many features that we do not require As a result, its implementation could prove to be time intensive and costly. In addition to the extra implementation features, the actual implementation is not as straightforward as Storm. In Spark, there exists a data structure known as resilient distributed dataset (RDD). RDDs are similar to topologies in Storm, but are not operational DAGs. They are considered to be immutable distributed collections of objects. The flow pattern is, however, similar to a graph. It is the job of the programmer to create transformations, and then Spark handles the DAG management, scheduling, and execution. This seems convenient, but the innate transformation functions Spark offers are not as flexible.

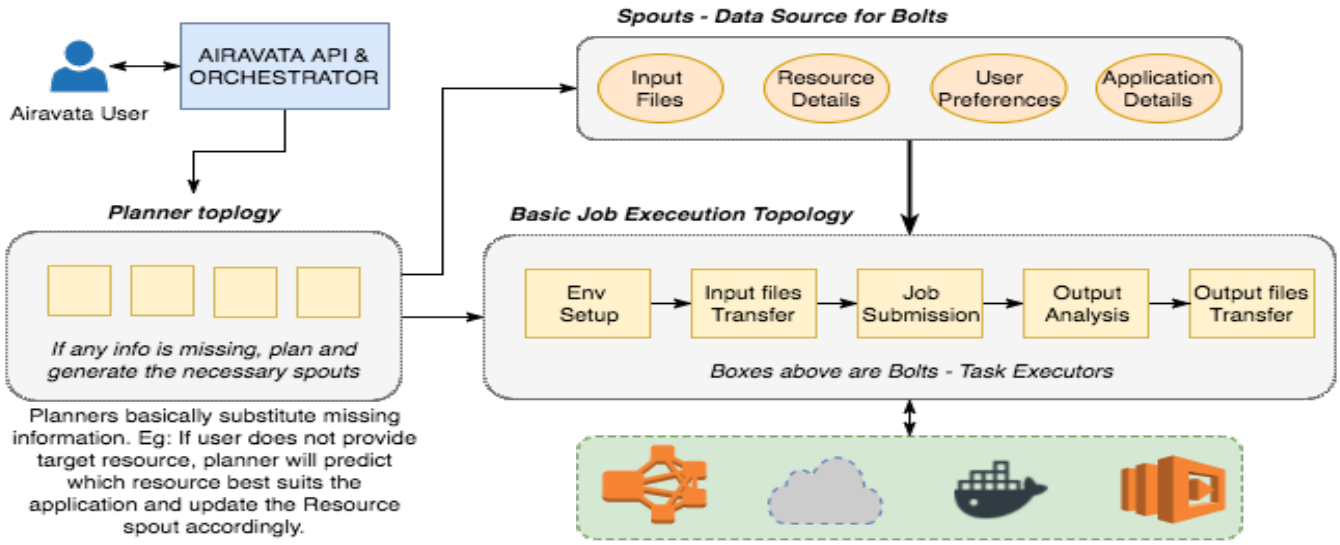

Fig. 3. Simple Airavata architectural overview with Apache Storm as task execution framework.

As a result, the programmer has to implement these execution specific features. Other than these drawbacks, Spark provides an excellent framework for Airavata jobs/tasks to be executed in a sequential manner.

IV. Apache Flink

Flink, similar to Storm, is a event streaming and processing framework. Its use case is closer to Storm than it is to Spark. Flink is very similar to Spark/Storm in that its framework provides high scalability, high fault tolerance, and relative ease of use. Flink utilizes pipelining and data parallelization to execute its workflow in a timely and organized manner. As a result, it is used extensively for running iterative machine learning algorithms in real time.

For our use case, we want to analyze Flinks use of DAGs and their execution. In Flink, DAGs are mapped to streaming dataflows. Each dataflow starts with a source of data input and ends with a sink. While the data is in the pipeline, calculations can be performed on it. These streams can be arranged into a DAG format to match the desired execution cycle.

Furthermore, Flink provides extensive support for running Storm and Spark code on Flink clusters. Though Flink has a different and distinct execution process, one can run Storm topologies on Flink clusters. The combination of Flink and Storm is an aspect we will need to evaluate. The control over functions in Flink is better than Spark. There exist more functions for our DAG management case. This could potentially be very positive as ease of use is a major concern for Airavata.

V. Conclusion

In this paper we summarized our efforts in exploring BigData frameworks as an implementation engine for Apache Airavata's distributed task execution needs. We digested the Apache Storm, Apache Spark and Apache Flink and their applicability. All three frameworks described above somewhat meet the criteria for our use case. Among the three, Apache Spark seems to be the most out of place framework. As of now, we have not built an operating model for the Spark use case. The Apache Storm and Flink models have been meeting expectations and performing as expected. In addition to this, error and exception handling is a major concern we need to address. Servers could potentially fail or individual components could shut down abruptly.

These frameworks need to provide good support for error and exception handling, which will eventually reduce the developer overhead and make development easy. Typical errors in scientific workflows relate to communication with remote resources such as supercomputers or clouds. All three frameworks provide basic exception handling mechanism out of the box, but as per our current literature understanding, they still lack the desired support. Though Storm seems to provide minimal exception handling support, it proves to be highly effective in virtually all other requirements. Flink provides better error handling, but proves difficult to manage custom topologies (DAGs). Implementing custom task executors in Flink, such as submitting a job to a remote cluster, is impossible. This is because Flinks transformation operators are limited to generic data processing functions such as MAP, REDUCE, JOIN, FILTER, etc. In contrast, Airavata needs complicated task executors, not just input data transformers. It is difficult to define custom complicated DAGs in Flink and Spark, which is relatively natural in Storm via topologies. We can also define complicated task executor logic in Storm via Bolts.

We are investigating the use of Flink in combination with Storm to amalgamate the best in both these frameworks benefiting Airavata. Considering the aforementioned analysis, Apache Storm seems to be a better suited candidate to replace the current GFac in Airavata to perform distributed Task

Execution.. By the time of the conference, we should have concrete comparison results.

VI. Related Work

As discussed in section 1, various facets of task execution challenges have been addressed over time and discussed extensively in distributed systems literature. Cyberinfrastructure projects have implemented these concepts in software systems to varying degree of successes. The larger vision of Apache Airavata is to leverage on existing systems to manage distributed task executions. In this paper, we evaluate existing big data frameworks to manage distributed workloads. While it is not appropriate at this point to do an 'apples to apples'comparison of these frameworks with other platforms, we briefly discuss the HTCondor [10] project which has tackled the problem of distributed task execution.

HTCondor offers a robust workload management system for compute-intensive jobs. It provides a job queueing mechanism, scheduling, priority scheme, monitoring, and resource management. HTCondor employs a matchmaking algorithm to schedule jobs on particular nodes [11]. It uses ClassAd mechanism for matching resource requests (jobs) with nodes [12]. Whenever a job is submitted to Condor, it states both the requirements and the preferences, such as required memory, name of the program to run, user who submitted the job, and a rank for the node that will run the job. Also, nodes advertise their capacities in terms of RAM, CPU type and speed, current load with other static and dynamic properties.

HTCondor can be used to build a highly-scalable Grid-style computing environment. It makes use of cutting edge Grid and cloud-based computing designs and protocols. HTCondor can be used to build Grid-style computing environments that cross administrative boundaries.

Acknowledgment

Apoorv Palkar is supported by the Google Summer of Code 2017 program.

References

[1] S. Marru, L. Gunathilake, C. Herath, P. Tangchaisin, M. Pierce, C. Mattmann, R. Singh, T. Gunarathne, E. Chinthaka, R. Gardler *et al.*, "Apache airavata: a framework for distributed applications and computational workflows," in *Proceedings of the 2011 ACM workshop on Gateway computing environments*. ACM, 2011, pp. 21–28.

[2] S. Marru, M. Pierce, S. Pamidighantam, and C. Wimalasena, "Apache airavata as a laboratory: architecture and case study for component-based gateway middleware," in *Proceedings of the 1st Workshop on The Science of Cyberinfrastructure: Research, Experience, Applications and Models*. ACM, 2015, pp. 19–26.

[3] M. E. Pierce, S. Marru, L. Gunathilake, D. K. Wijeratne, R. Singh, C. Wimalasena, S. Ratnayaka, and S. Pamidighantam, "Apache airavata: design and directions of a science gateway framework," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 16, pp. 4282–4291, 2015.

[4] S. Perera, S. Marru, and C. Herath, "Workflow infrastructure for multi-scale science gateways," in *TeraGrid Conference*, 2008.

[5] T. L. Casavant and J. G. Kuhl, "A taxonomy of scheduling in general-purpose distributed computing systems," *IEEE Transactions on software engineering*, vol. 14, no. 2, pp. 141–154, 1988.

[6] "Apache mesos," http://mesos.apache.org, 2017, accessed: 2017-03-13.

[7] "Apache storm," http://storm.apache.org/, 2017, accessed: 2017-03-13.

[8] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets." *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

[9] "Apache flink," http://flink.apache.org, 2017, accessed: 2017-03-13.

[10] T. Tannenbaum, D. Wright, K. Miller, and M. Livny, "Condor: a distributed job scheduler," in *Beowulf cluster computing with Linux*. MIT press, 2001, pp. 307–350.

[11] N. Coleman, "An implementation of matchmaking analysis in condor," *Masters' Project report, University of Wisconsin, Madison*, 2001.

[12] N. Coleman, R. Raman, M. Livny, and M. Solomon, "Distributed policy management and comprehension with classified advertisements," Technical Report UW-CS-TR-1481, University of Wisconsin-Madison Computer Sciences Department, Tech. Rep., 2003.