UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
UNIDADE ACADÊMICA DE SISTEMAS E COMPUTAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

RAONI MATOS SMANEOTO

A RESILIENT AND CLOUD-BASED BATCH PROCESSING SYSTEM FOR HIGH PERFORMANCE COMPUTING

CAMPINA GRANDE - PB

2020

RAONI MATOS SMANEOTO

A RESILIENT AND CLOUD-BASED BATCH PROCESSING SYSTEM FOR HIGH PERFORMANCE COMPUTING

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

Orientadores:
Professor Dr. Francisco Vilar Brasileiro.
Professor Dr. Thiago Emmanuel Pereira da Cunha Silva.

CAMPINA GRANDE - PB

2020

S635r Smaneoto, Raoni Matos.
A resilient and cloud-based batch processing system for high performance computing. / Raoni Matos Smaneoto. – 2020.

12 f.

Orientadores: Prof. Dr. Francisco Vilar Brasileiro; Professor Dr. Thyago Emmanuel Pereira da Cunha Silva.
Trabalho de Conclusão de Curso - Artigo (Curso de Bacharelado em Ciência da Computação) - Universidade Federal de Campina Grande; Centro de Engenharia Elétrica e Informática.

1. Batch systems. 2. High performance computing. 3. Cloud elasticity. 4. Cloud computing. I. Brasileiro, Francisco Vilar. II. Silva, Thiago Emmanuel Pereira da Cunha. III. Título.

CDU:004(045)

Elaboração da Ficha Catalográfica:

Johnny Rodrigues Barbosa
Bibliotecário-Documentalista
CRB-15/626

RAONI MATOS SMANEOTO

A RESILIENT AND CLOUD-BASED BATCH PROCESSING SYSTEM FOR HIGH PERFORMANCE COMPUTING

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

BANCA EXAMINADORA:

Professor Dr. Francisco Vilar Brasileiro
Orientador – UASC/CEEI/UFCG

Professor Dr. Thiago Emmanuel Pereira da Cunha Silva
Orientador – UASC/CEEI/UFCG

Professor Dr. Adalberto Cajueiro de Farias
Examinador – UASC/CEEI/UFCG

Professor Dr. Tiago Lima Massoni
Disciplina TCC – UASC/CEEI/UFCG

Trabalho aprovado em: 2020.

CAMPINA GRANDE - PB

A Resilient and Cloud-Based Batch Processing System for High Performance Computing

Raoni Matos Smaneoto
Universidade Federal de Campina Grande
raoni.smaneoto@ccc.ufcg.edu.br

Francisco Vilar Brasileiro
Universidade Federal de Campina Grande
fubica@computacao.ufcg.edu.br

Thiago Emmanuel Pereira da Cunha Silva
Universidade Federal de Campina Grande
temmanuel@computacao.ufcg.edu.br

ABSTRACT

Since computers were able to execute more than one program at a time, the batch systems became usual. In this context, many batch applications require a high level of processing capacity, which leads us to high performance computing. This approach has been used for long, mainly for scientific purposes. It is common that the conventional environments for *HPC*, which are local clusters and supercomputers, provide a command line interface for the users to enable them to run their applications in nodes connected through high-speed networks. However, high-speed networks might not be useful in single-node applications, becoming a waste point. Besides, despite the old users being used to the command line, it may frustrate newcomers. That's why we propose, with this work, a *HPC* supported system that takes advantage of some cloud's features to minimize the cost and the waiting time, while focusing on the user experience.

Keywords

Batch, cloud, high performance computing, user experience.

1. INTRODUCTION

A batch application or job is a group of tasks and their associated input data that can be executed without user interaction. Batch processing systems have become useful since computers were able to execute more than one program at a time. These systems can run a stream of jobs whose processing order can follow several strategies. A very common way is to follow a First-In-First-Out (FIFO) policy, in such a way that the first job to enter the system is the first job to be processed. A job can be scheduled to run as soon as the system has the required resources available. These resources compose the system's environment, that can be a group of nodes, for example, and that has a huge influence in the system performance, because the more resources the system has, the less the jobs wait to be executed.

The usage of batch applications is very wide, varying from complex scientific simulations, to simple task automation, like printing a collection of documents or collecting logs. Each batch job is composed of a set of tasks. The dependency relationship of these tasks characterizes the job's structure. Thereby, batch applications can range from complex ones, whose structures are represented by *direct acyclic graphs* (DAG), to simple ones comprising a set of independent tasks, known as bag-of-tasks applications. Depending on the level of complexity of the application, its tasks might interact frequently, which leads to the need of using high-speed networks, in order to avoid delay in the progress of each task.

The notable utility of batch systems has made many big companies, and even some open source initiatives, invest some effort to develop such systems, each one with its particularity. Some of these systems are very famous nowadays, like *celery* [10], *nomad* [6] and *apache mesos* [7].

Due to the non-interactive characteristic of batch applications, the systems above, and any other batch system, do not provide a way for the user to interact with applications during their runtime. It restricts the user interactions to deployment and execution phases interactions. During the deployment phase, the users' goal is to let the application ready for execution. In general, this is not a simple process. Some systems require the user to ask the system's administrator to deploy the application, others make the environment accessible, so the user himself/herself is able to manually install the application in the system, and just a few provide a friendly interface to deploy the application [15]. When it comes to the execution phase, usually, the user's interactions are composed of three main steps: stage in of input data, processing and stage out of results. In the stage in step all the data the application needs to be executed is provided, in the processing step the application execution is triggered and in the stage out step the execution results are retrieved. There are a few ways to provide interfaces for such steps. Some systems, for example, expose a *RESTFul* API (Application Programming Interface),

others provide a Command Line Interface (CLI) client, and there are still some of them that do not provide a specific interface for that, which makes the user experience more complex.

Some batch applications need a significant amount of compute resources to be executed, they are known as High Performance Computing (HPC) applications. Local clusters and supercomputers have been for long the main environments used by these applications. Nowadays, though, cloud providers are also supporting these applications. The natural elasticity characteristic of clouds has a big impact on this scenario, because it may significantly decrease the wait time of the applications' tasks, and in some situations it can save resources. Beyond that, the pricing follows the pattern of other cloud services; the user pays for what he or she uses. These aspects might take the cloud as a good alternative for some *HPC* applications.

Thereby, the promising future of *HPC* in the cloud encouraged us to propose an architecture of a batch processing system for cloud-based infrastructures, which covers this scenario and strengthens this approach. For this, the system addresses some requirements, in order to take advantage of the cloud main features, when compared to convencional *HPC* environments, like autoscaling, optimization of resource usage and high resilience and availability, while also taking into account the user experience.

2. BACKGROUND

This section goal is to put some matters into context, so they can be discussed more clearly in the following sections. The first topic covered consists in the user experience in conventional *HPC* infrastructures, followed by cloud elasticity, and lastly resilience and availability in cloud computing.

2.1 User Experience in Conventional *HPC* Infrastructures

To improve the user experience in *HPC* infrastructures, many solutions have been designed: web interfaces, like *Open OnDemand* that provides an easy way for systems' administrators to provide web access to their *HPC* resources; desktop clients, like Eclipse Parallel Tools Platform; and science gateways, which consist in domain-specific web interfaces. However, most *HPC* infrastructures have the command line as their primary user interface and despite part of the users being used to it, most of them are not comfortable with such an interface [1], [5].

Many of the HPC users come from the scientific community. Although some of them might be used to command line interfaces, many face difficulties that come from unfriendly interfaces. Thus, besides avoiding common errors that come from unfriendly interfaces, a web interface might provide other features, like the modern authentication, and ease cross-domain science by integrating information from multiple sources and making them accessible through the same interface.

2.2 Cloud Elasticity

Cloud elasticity is a cloud feature that enables adding and removing resources on demand. There are two possible types of elasticity: vertical and horizontal. The vertical elasticity stands for the ability to adapt some computing resources properties, like the quantity of cpu cores, the amount of network bandwidth, etc. Horizontal elasticity consists in the ability to adapt the number of computing resources available, increasing or decreasing as needed. Ideally, both of them should be automatically triggered by changes in the current workload demand.

Another important aspect of cloud elasticity is that it always looks for optimization, meaning that it aims to match the amount of resources available with the amount of resources required by the workload at a given time. The pricing of these resources follows a pay-as-you-go policy, which means that the user only pays for what he or she is using at a time.

2.3 Resilience and Availability in Cloud Computing

Resilience and availability are two essential properties that aim at the correct functioning of systems, which might maintain the users' confidence and prevent possible revenue losses. Some authors define resilience as a measure of fault tolerance [12], which stands for the ability of a system to work properly in the presence of some failures. Availability can be understood as the percentage of time a system is able to operate as expected; here, the recovery time is also important, because the longer a system takes to recover, the longer it will take to be available again.

When it comes to these properties, cloud computing plays an important role, because of two reasons: cloud providers usually invest effort in order to develop services that either make their infrastructure more available and resilient or provide the users with these properties; some mechanisms which clouds are based on, like virtualization, ease some common mechanisms that improve resilience and availability.

Resilience and availability have always been a concern to cloud providers. Amazon, for example, has developed many services with this goal. For example, AWS autoscaling is a service that integrates to a system's architecture the ability of automatically scale its services based on current demand [6]; Amazon CloudWatch is a monitoring service that ease anomalous behaviour detection [2]; Amazon elastic load balancer [3] handles entry and removal of nodes automatically, and deals with sudden network traffic changes etc. Besides, some mechanisms regarding

fault tolerance in distributed systems that have a huge influence in resilience and availability, like replication and scalability, fit very well in cloud infrastructures. The ease in getting new computing resources in clouds, mainly because of virtualization, and that makes the resources easily changeable, is very important. For example, if one node of an application is not operating correctly, it can be easily destroyed and replaced by another, that can be even in another physical place, in a transparent way for the users, if there were more than one node running the application. The elasticity is also important here, because it gives the users the ability of scaling their application automatically based on the current workload of the applications, preserving their health and making them more resilient and available.

3. REQUIREMENTS

When a new proposal of a product already consolidated in the community arises, there are a few requirements it has to follow to be considered by the community members. These requirements might guarantee that the product is at least as good as other existent solutions, and even better in some scenarios. In the context of this work, the requirements' goals are to show to the community that *HPC* applications can be executed in the cloud, with the possible benefits of minimizing cost, shortening the application waiting time, resilience and availability, all of that without compromising the user experience.

3.1 Friendly User Experience

Once *HPC* is largely used for science purposes, it is required that the scientists have a satisfactory experience in such systems. Most of the users' experience comes from the interface they are going to use to interact with the system. First we have to take into account that currently many users are used to the command line interface, which is one of the most used in this context. Changing this experience might be frustrating. On the other hand, the command line interface is not that friendly, which makes us believe that it can lead to many user-side errors and consequently decrease the productivity of a team, which can also be reinforced by the fact that frequently only some of the teams' members are comfortable with this kind of interface.

Therefore, we have to consider both the current users and the newcomers, meaning that our interface must support a command line client, aiming to reach the current users, and also more friendly ones, which might result in less errors and could even provide features that are not possible through the command line, resulting in more productivity and engagement, mainly by the new users. Thus, it is important that the proposed architecture aims to provide a good user experience regardless of their knowledge about computing systems.

3.2 Minimize Wait Time

The elasticity can be an important player to the system's performance. The ease it provides in getting and setting up new resources might result in a shorter, or even none waiting time. Thus, each time there is a new application waiting in the queue, the system can provide new resources to run the application, if the required ones are not available. This is not possible in local clusters, for example. In order to boost them, one needs to buy the new resources and set them up manually, which might take a huge amount of time. A shorter application waiting time is very feasible in the cloud infrastructure approach.

3.3 Minimize Cost

Supercomputers and local clusters, which are one of the most used infrastructure for *HPC*, have been designed to support huge workloads and intensive task communication. For this reason, they usually have high-speed and consequently high-cost networks. However, a considerable part of *HPC* systems workloads are single node applications and this scenario does not require high-speed networks, once there is no intensive communication between nodes [6]. Thus, there is a waste of resources, specifically of network resources, in the conventional *HPC* infrastructures.

Besides the network issue, there is still another point of wasting regarding possible idle resources. As the system has a shorter wait time as a requirement and because this requirement is based on the scale out part of cloud elasticity, it is possible that, in a specific moment in time, some available compute resources, that once had some job to do, are idle.

Therefore, it is important that the system be aware of these two issues in order to avoid unused resources, minimizing the cost.

3.4 Deal With the Trade-off: Minimize Wait Time Versus Minimize Cost

Scaling the resources out is the main alternative to handle with increasing workload and to minimize the wait time. On the other hand, scaling them in is the main alternative to minimize the cost and avoid idle resources. Thus, because the system is worried with both directions, the protocols it follows must be as precise as possible to make it clear whether the resources need to be scaled in or out, depending on the demand.

3.5 Resilience and Availability

Resilience and availability are two properties that fit very well in cloud infrastructures, but in order to take the maximum advantage of this, the proposed system's architecture must support it. The components design and the way they interact must allow that some mechanisms of fault tolerance, like replication and scalability, are applicable. Thus, by taking advantage of the cloud infrastructure and the way it is built, the system can reach the

availability desired by the users and provide a good user experience even upon failures of some components.

4. SOLUTION

In this section we detail the proposed solution. It is composed of four subsections: the first one focuses on pointing out some of the important decisions we have made regarding the architecture; the second one describes the architecture and its main components; the third one describes the protocols the system must follow in order to meet the expected functioning and the requirements. These protocols explain when and how the components communicate with each other; in the fourth subsection we explain why the proposed solution meets the requirements; lastly we present a proof of concept, illustrating how the mais parts of the proposed system could be implemented.

4.1 Rationale

Before delving into the details of the architecture's description, we discuss some of the design decisions taken. Starting from a broader perspective, we decided that the system should have RESTful interfaces as the gateway to its core, because we believe they are easy to integrate with other clients. Besides, the system works in a pull-based communication style, meaning that the core component works in a passive way, answering the incoming requests from the other components, instead of requesting them. This is important because the core component does not need to know the others, and can work completely independent. More importantly, this helps deployment by concentrating the configuration of firewall rules at the core component side.

Focusing in the core component, we have also taken some important decisions. We decided that the system must support multiple queues. The queues are the structures where jobs live. The first interesting feature enabled by multiple queues comes from another core decision, it consists in customized scheduling policy per queue, which, in other words, allows that each queue has a specific way to choose the next job to execute. Another feature it enables is a semantic split of jobs; one might create multiple queues and insert the jobs to them based on the semantics attached to each queue, thus, jobs from different contexts do not interfere in each other's scheduling. HPC users are already used with the existence of multiple queues in typical HPC infrastructures, however, in the system that we propose, they add extra flexibility.

4.2 Architecture

The proposed architecture's main goal is to fulfill the requirements previously described. It is composed of three main parts: the Server, the Workers and the ResourcesManager. The Server is the core of the system, it exposes a RESTful API and its components interact with each other aiming at managing the execution of jobs and providing their results properly. The Workers are responsible for executing the tasks and report their results; periodically, the worker also reports its progress. The communication between the workers and the server is pull-based, such that it is always triggered by a worker. The last part is the ResourcesManager. It interacts with the server in order to check if the current resource availability matches the current workload. Depending on the scenario, it can either scale the system's resources out, if there are not enough resources to handle the current workload, or scale them in, if there are any idle resources.

This architecture is summarized in Fig. 1. We describe the key components below, to explain in more details how they interact to achieve the system's goal in the protocols subsection.

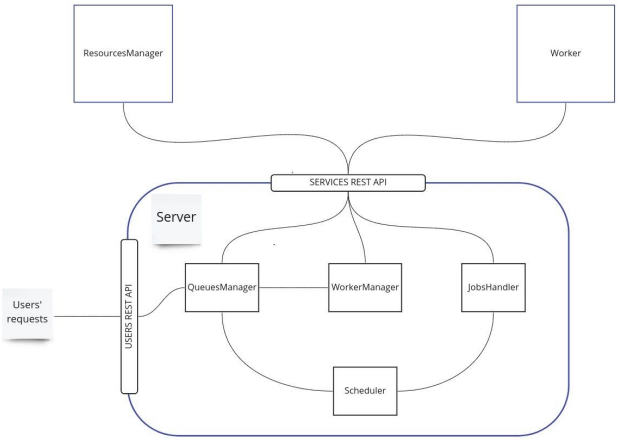

Fig. 1: Proposed architecture big picture

4.2.1 Queue

This is an abstraction that works like a jobs' buffer to the system. The users can create queues to keep their jobs, according to their semantics. For each queue, a Scheduler is assigned, and for this reason, the scheduling policy can change from queue to queue.

4.2.2 Worker

A worker is a component responsible for executing tasks and providing their outputs. It runs in a node previously setup by the ResourcesManager. To do its job, it communicates with the server through the REST API. This communication occurs in the joining, the task request and the task results reporting processes.

4.2.3 REST Apis

The REST APIS are the server's gateway to both the users and the others components. In order to serve the users' requests, the API uses the QueuesManager. When it comes to the other components' requests, the API might use directly the QueuesManager, the WorkerManager and the JobsHandler.

4.2.4 WorkerManager

This entity authenticates the incoming workers and decides which queue they will be attached to. To do so, it either uses a tag in the worker's configuration that indicates the queue it must be attached to, or it uses the QueuesManager to get information about the queues' workload and chooses one. Thus, the component can

make the system able to balance the load of the queues, by allocating more workers to those that have more jobs than others, or even to prioritize a specific queue that may be more important at the moment. The WorkerManager keeps the state of the workers available per queue.

4.2.5 QueuesManager

The QueuesManager is the entity that has knowledge about the queues. As such, it is responsible for all the operations that concern queues. These operations include: queues' creation; queues' retrieval; jobs' insertion; and the attachment of a JobsHandler to each queue.

4.2.6 Scheduler

The Scheduler has a buffer to keep some ready-to-run tasks. Its job is to feed this buffer, when it is convenient, and to provide tasks to workers. The Scheduler works according to its policy. It can, for example, schedule an available task to a requesting worker based on the task deadline. The task's requirements are also important and considered before the dispatching; they must match with the worker's configuration, ensuring that the worker is able to execute that task, from a resource viewpoint.

4.2.7 JobsHandler

The responsibility of this component is to manage the jobs' state. To do so, it first gets the jobs from the queue and extracts their tasks. Then, the JobsHandler split the PENDING tasks, which are ready-to-run, to be able to send them to the scheduler, when it asks for them. The JobsHandler also resolves report requests, in which it can change the state of the reported task and of its job, if all tasks of this job have already finished. There is only one more thing the component must be aware of. Some tasks can never end, which may lead to a never ending job as well. To deal with this problem, the JobsHandler assigns a *report interval* time to each task it has sent to the scheduler. The worker needs to report the current task's status after each *report interval* time. The JobsHandler then verifies periodically if there is a *RUNNING* task that has not been reported for a period greater than its *report interval.* If that is the case, the task is sent to the scheduler again and its state set to PENDING. Thus, every task is able to reach a final state (*FINISHED* or *FAILED*), preventing the previous problem.

4.2.8 ResourcesManager

Just like the workers, it is a separated component. As described in the architecture general description and as its name suggests this entity aims to solve a possible unbalance between the resources available and the system's workload. The ResourcesManager communicates with the server to decide whether workers need to be added or removed. The scaling is performed by interacting with resource providers. The component stores state about which providers are available and which workers have been acquired with each provider. It is also in charge of setting up the worker after it is provisioned. This autoscaling feature brings efficiency and resilience to the whole system, once it is totally dependent on the workers.

4.3 Protocols

Once the architecture's main components have been described, it is important to detail how they interact with each other in order to guarantee the correct functioning of the system. Thus, this section describes the core protocols of interaction between the components, which include the balance checking protocol, resource creation protocol, resource removal protocol, worker's get task protocol and the task execution protocol.

4.3.1 Balance Checking Protocol

This protocol describes the process by which the ResourcesManager checks the current balance between resources available and pending workload, it is needed to make the system able to adapt the resources according to the current demand. This process is triggered periodically. By checking this balance, the ResourcesManager can trigger either the resource creation process or the resource removal process, whose protocols are described later. This protocol includes the following steps:

1. ResourcesManager requests the current state of the system: The ResourcesManager sends a request to the server to retrieve the current system's state. This request is signed with the ResourcesManager's private key; the server is able to verify this signature, once it keeps the ResourcesManager's public key, placed in deployment time. Once the signature is verified, the server uses the QueuesManager to get information about each queue's workload and uses the WorkerManager to get information about each queue's available resources. Then, the server summarizes this information generating a snapshot and returning it to the ResourcesManager.

2. ResourcesManager checks the state of the system: Now, with the snapshot in hands, the ResourcesManager checks the possible unbalance between the queues' workload and the available resources. There are three possibilities per queue here: in the first one, the ResourcesManager figures out that the current number of ready-to-run tasks in the queue is greater than the number of available resources. In this situation, the ResourcesManager triggers the resource creation process. In the second possibility, the ResourcesManager figures out that there are more resources available than workload, and triggers the resource removal process. Last, the ResourcesManager does not find any unbalance between the queue's workload and the available resources and does not trigger any other process.

4.3.2 Resource Creation Protocol

This protocol describes the process by which a worker is added in the system. It is the result of interaction between the ResourcesManager and the Server. This interaction is also focused in preventing malicious workers from joining the system, and as such it is important to highlight that the ResourcesManager's public key is pasted in the server in deployment time. It includes the following steps:

1. ResourcesManager retrieves available IDs from the server: The server keeps an *allow list* created in deployment time. This *allow list* contains the IDs available to new workers joining the system. In the first step of this protocol, the ResourcesManager sends a request signed with its private key to the server, requesting an available id from the *allow list*. The server checks if the signature is valid and, if so, it returns the first available id of the *allow list* to the ResourcesManager.

2. ResourcesManager creates a resource along with a resource provider: With a valid id in hands, the ResourcesManager is able to instantiate the worker node. It chooses one of the available resource providers and requests a new computing resource. If the request fails, the ResourcesManager chooses another resources provider. Once the resource is ready, the ResourcesManager stores the resource id along with the worker id and the responsible resource provider.

3. ResourcesManager starts the worker: After the computing resource is ready, the ResourcesManager sends a startup script along with a configuration file to the resource and executes it. The configuration file contains, besides other fields, the worker's id and the id of the queue to which the worker will be attached. The script setups the machine and starts the worker's process. Now, the worker is ready to run.

4.3.3 Resource Removal Protocol

This protocol describes the process by which a worker is removed from the system. It is triggered by the ResourcesManager and includes the following steps:

1. ResourcesManager requests worker removal to the server: Once the ResourcesManager has noticed that there are more resources than ready-to-run tasks in a specific queue, it requests the server to remove that worker informing the queue's id and the worker's id.

2. ResourcesManager requests the resources provider to remove the resource: At this point, the server does not count with the worker anymore. The ResourcesManager then requests the resource deletion to the resource provider informing the resource id, both the resources provider and the resource id have been stored at the creation time.

4.3.4 Worker Get Task Protocol

This protocol describes the process by which workers get a task to execute. We also included a sequence diagram in the Fig. 2, to help the understanding. These are the steps:

1. Worker joins the server: First, the worker joins the server by sending a request that contains its public key, its configuration and its id. If the id is not a valid one, the server returns a forbidden error. Otherwise, the WorkerManager will check for which queue the worker is going to be assigned to, save this association and return the queue's id and a token to the worker. The queue can be chosen based on the worker's configuration, a predetermined scheduling policy or a tag that explicitly informs which queue that worker has to be attached. The token is valid during a time interval known as lease time. With the queue's id and the token in hand, the worker is able to get tasks from that queue until the lease expires. After expiration, the worker must ask for a new one to resume operating. The server is allowed to choose another queue to be associated with the new token.

2. Worker requests a task: The worker requests a task to the server informing its id, its queue id and its token. The request is signed with the worker's private key. The server verifies the token and, if it is invalid, an error informing this case is returned. If no problem has been found in the token and the signature, the request proceeds.

3. Task scheduling process: After the request validation, the server needs to choose a task from a job that lives in the specified queue. The Scheduler chooses based on the scheduling policy associated with the queue and dispatches the task to the worker as a response from the request and changes the task's state to RUNNING.

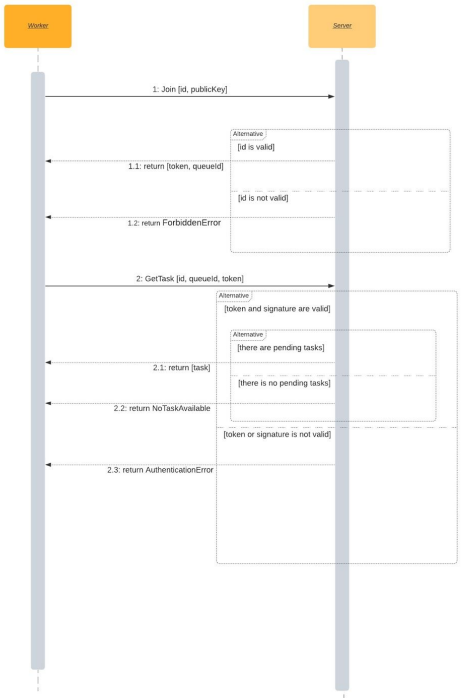

Fig. 2: Sequence diagram of get task protocol

4.3.5 Task Execution Protocol

A task is the executable piece of the system. The task execution protocol happens during runtime and involves the worker and the server. It includes the following step:

1. Worker reports execution status: When the task is dispatched to the worker, a *report interval* is informed. Its meaning stands for the time interval in which the worker must report the task status for the server. The report may be the current task's state followed, optionally, by the percentage of the task that has been already executed. This progress information allows the implementation of simple fault tolerance mechanisms. For instance, if a task has been dispatched with a *report interval* set for t and in an interval of $t + 1$ no report about that task arrives, the server can assume that the worker has failed. Once the server detects the failure of a worker, it can dispatch the task to another worker. It also allows the server to optimize the task execution in some situations. For example, with this mechanism, the server can spot slow tasks, which might lead it to replicate the task and send it to another worker. Whoever finishes the task first now is the winner. By doing so, in the worst case, the task will be executed as slowly as it would be if no replication had taken place.

   When the worker finishes the task execution, it reports the results to the server. The report now has a task final state (finished or failed). When the server receives the report it changes the task's state to the received one and the worker is free to execute another task.

4.4 Requirements Meeting

The first requirement, satisfactory user's experience, describes the need of comfortable interfaces to the users. As part of the old *HPC* systems users are used to command line interfaces, we set it as mandatory to support this kind of client in the system. On the other hand, we believe that simpler clients, like web clients, would result in less user-side errors, increasing the engagement, which makes us also consider it mandatory to support them. Thus, as we described in the architecture subsection and complemented in the protocols subsection, the system provides a RESTFul API. The API supports both command line based clients and web based clients, because they are able to reach the server API by sending HTTP requests.

The requirement of minimizing wait time is also addressed by the proposed solution. The balance checking protocol along with the resource creation protocol describes the process by which the ResourcesManager provides new resources. The supply of new resources happens as soon as the ResourcesManager realizes that there are more tasks than workers in a queue, which in turn decreases the waiting time for the waiting applications. However, the ResourcesManager also worries about idle resources. When it realizes that there are more workers than tasks in a queue, it removes the idle resources, decreasing the cost.

As the reader can see, the system worries about both minimizing wait time and minimizing cost requirements. By checking the balance between workload and available resources periodically, the ResourcesManager is able to provide new resources or to remove the idle ones based on the current demand per queue. Each one of these actions are taken carefully after the balancing checking process is done, which avoids unwanted scenarios and allows us to say that the system deals with the trade-off between minimizing wait time and minimizing cost satisfactorily.

When it comes to resilience and availability, there are some important aspects of the system to highlight. The workers are very important to the system and because they are decoupled from the server, it is easy to create or remove them. Thus, a bad functioning worker may not be an issue to the system, once it has mechanisms to replace that worker and run again the tasks it eventually retrieved to execution, as described in the task

execution protocol. Besides, in case of failure of the ResourcesManager, because the WorkerManager has a scheduling ability, the system can still work with the current available workers. For that, the WorkerManager gets the responsibility of deciding for which queue a joining worker will be attached, here the joining workers are mainly the already existent ones whose lease time have expired. These fault-tolerant aspects can prevent the system from having tasks that will not ever be completed, and can prevent from bad functioning in case of failure of either the workers or the ResourcesManager, which increases its resilience. The server's components have been designed with their responsibilities well defined. Also, they are as decoupled as possible from each other. These characteristics alongside some adjustments, like API replication to each one, may allow them to work separately and to be replicated, which isolates failures and increases both resilience and availability.

4.5 Proof of Concept

A proof of concept implementation of the system proposed in this manuscript is under way. So far, it contemplates the worker and some of the server's features, that are described in the following. We used some technologies such as golang, docker and shell script to develop it. The ResourcesManager is yet to be implemented. We discuss how this component and the missing features can be implemented. This discussion gives some indication on the feasibility of our proposal.

4.5.1 Worker

The worker's implementation [16], starts in a loop in which it tries to get tasks. If the get task request fails because authorization's issue, the worker joins the server and tries to get the task again. With the task in hand the worker is able to execute it. In the execution phase, the worker instantiates a task executor, it is for instance a docker client, but could be another driver, and triggers the execution in another thread. It also creates a ticker which fires an alarm each time the task interval is reached. In case of the task execution is done, the executor also fires an alarm. The worker waits for these two alarms and reports the task progress each time one of them is fired.

4.5.2 Server

The server [11], exposes its apis. Currently, the join and the get task api handlers are implemented. The join api handler invokes the WorkerManager to add the worker. The WorkerManager first verifies if the joining worker's ID is available in the allow list; then it saves the worker's public key in such a way that it can be retrieved by its ID; in the third step, the WorkerManager checks if the signature is valid and generates a token; lastly it retrieves the queues by requesting the QueuesManager and chooses the one to which the worker will be attached, keeping state of this. After that, the api handler returns the token and the queue's ID.

In the get task api handler, the server authenticates the worker checking the request signature; then it authorizes the worker by checking its token; and lastly, it retrieves the queue's scheduler from the QueuesManager and asks the scheduler for a task, returning it to the worker

Despite the api report handler hasn't been implemented, there is a routine in the jobs handler responsible for that, it updates the task's progress and state, and persists it.

Another important action here stands for the JobsHandler starting point. It started before the api got exposed. When started, the JobsHandler starts three other threads. The first is responsible for collecting ready-to-run tasks from the queues and for keeping them to send to the scheduler when it asks for. The second checks for never ending tasks by looking for tasks whose last update time plus its report interval is greater then the current time. For each one this is true, the JobsHandler set the state as ready-to-run again. The third keeps tracking of the jobs' state, updating them every time all their tasks are done.

4.5.3 Credits

The implementation described above has been conducted by two main developers, Raoni Matos Smaneoto and Wesley Henrique Araújo Monte [11], [16]. While Wesley is responsible for the implementation of the join operation, the worker's authorization and authentication in the server side, Raoni focused on the worker's implementation, the server's get task operation and the JobsHandler's implementation.

4.5.4 Yet to Come

The ResourcesManager implementation is yet to come. Once the server already keeps the states needed by the ResourcesManager, which consists in the queues' load and the workers attached to them, the implementation that is to come consists mainly in the ResourcesManager itself. This component will be the component responsible for interacting with the resources providers, and as such, it needs to provide a way of easy integration. A feasible one consists in adding a configuration file for each one of the resources providers. The configuration file contains information about how to authenticate with the provider and the endpoints the ResourcesManager needs to hit to reach each one of its goals. The ResourcesManager would interact with the resources providers each time it finds out an unbalance after analysing the snapshot provided by the server, keeping state about the resources it has created and the workers' configuration attached to them.

5. CONCLUSION

With the proposed solution, we provide an alternative to the conventional *HPC* systems, focusing on the users' experience, resilience, performance and cost. Thus, the users can work with *HPC*, interacting with an interface that is able to support both command line clients, which pleases the users that are used to the

conventional systems, and modern clients, which pleases newcomers users who are not comfortable with command line interfaces. Besides, the system ability of dealing with failures, by isolating them, improves the system's resilience, making some of the main components capable of working regardless of the others.

When it comes to performance, the system leaves nothing to be desired, it takes advantage of the cloud's elasticity and provides resources everytime that the load is greater than the resources available per queue. Minimizing the cost is also an issue the system cares about, the ResourcesManager removes idle resources as soon as it realizes that there are more resources available than ready-to-run tasks per queue.

Therefore, the cloud infrastructure has been a key player in providing such characteristics and has shown itself very promisor to the brief future of high performance computing. We conclude that the proposed system is an important initiative in bringing high performance computing to the cloud infrastructure.

6. ACKNOWLEDGMENTS

I thank my parents for always believing in me and for always being there. I thank my brother, Tiaraju, who also always believed in me, who is always present regardless of the circumstances and who presented computer science for me. I thank my cousin Kauê for all the support he gives me. I thank all my family for being so lovely with me. I thank all my friends, Gustavo Ribeiro, Emerson, Ignácio, Isabelle, Erilânia, Ana Beatriz, Leonara, Gustavo Melo, Caíque, Wesley and Vinícius. I thank my advisors Francisco and Thiago for their patience, wisdom and all opportunities they have given me during the graduation.

7. REFERENCES

[1] Alameda, J. Supporting Modern User Interfaces for High Performance Computing. Available at: https://cpb-us-w2.wpmucdn.com/sites.udel.edu/dist/6/8980/files/2019/03/Supporting-Modern-User-Interfaces-for-High-Performance-Computing.pdf

[2] Amazon. Amazon CloudWatch. Available at: https://aws.amazon.com/pt/cloudwatch/

[3] Amazon. Elastic Load Balancing. Available at: https://aws.amazon.com/pt/elasticloadbalancing/?sc_channel=PS&sc_campaign=acquisition_BR&sc_publisher=google&sc_medium=english_load_balancing_b&sc_content=aws_load_balancer_e&sc_detail=aws%20load%20balancer&sc_category=load_balancing&sc_segment=159751609434&sc_matchtype=e&sc_country=BR&s_kwcid=AL!4422!3!159751609434!e!!g!!aws%20load%20balancer&ef_id=CjwKCAjw_sn8BRBrEiwAnUGJDq8hFkdUifcNlaD3rCw1DancGGMKmqTJvTl2twP2nl0wL9YA1KAELhoCrzUQAvD_BwE:G:s&s_kwcid=AL!4422!3!159751609434!e!!g!!aws%20load%20balancer

[4] Barbosa, J. G. and Moreira, B. Dynamic scheduling of a batch of parallel task jobs on heterogeneous clusters. Available at: https://www.sciencedirect.com/science/article/pii/S0167819111000020

[5] Cholia, S., Skinner, D. and Boverhof, J. NEWT: A RESTful Service for Building High Performance Computing Web Applications. Available at: https://www.researchgate.net/publication/224208203_NEWT_A_RESTful_service_for_building_High_Performance_Computing_web_applications

[6] Freeh, V. W., Lowenthal, D. K, Pan, F., Kappiah, N., Springer, R., Rountree, B. L. and Femal, M. E. Analyzing the Energy-Time Trade-Off in High-Performance Computing Applications, (June 2007)

[7] HASHICORP. Introduction to Nomad. <https://www.nomadproject.io/intro>. Last access: April 30th, 2020

[8] Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A. Joseph, A. D., Katz, R., Shenker, S., Stoica, I. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. Available at: https://people.eecs.berkeley.edu/~alig/papers/mesos.pdf

[9] Jeff, B. New AWS Auto Scaling – Unified Scaling For Your Cloud Applications, 2018. Available at: https://aws.amazon.com/pt/blogs/aws/aws-auto-scaling-unified-scaling-for-your-cloud-applications/

[10] Milne, L., Lindner, D., Bayer, M., Husmeier, D., McGuire, G., Marshall, D. F. and Wright F. TOPALi v2: a rich graphical interface for evolutionary analyses of multiple alignments on HPC clusters and multi-core desktops. Available at: https://academic.oup.com/bioinformatics/article/25/1/126/302670

[11] Server's implementation. Available at: https://github.com/ufcg-lsd/arrebol-pb/tree/develop

[12] Solem, A. Celery - Distributed Task Queue. Available in: <https://docs.celeryproject.org/en/stable/> . Last access: April 30th, 2020

[13] Weik M.H. (2000) batch processing. In: Computer Science and Communications Dictionary. Springer, Boston, MA. Available at: https://doi.org/10.1007/1-4020-0613-6_1408

[14] Welsh, T. Perspectives on Resilience in Cloud Computing: Review and Trends. Available at: http://eprints.staffs.ac.uk/4420/1/8.pdf

[15] Wong, A. K. L. and Goscinski, A. M. A unified framework for the deployment, exposure and access of HPC applications as services in clouds

[16] Worker's implementation. Available at: https://github.com/ufcg-lsd/arrebol-pb-worker