A lightweight python based cluster and workload manager

Ali Hassani

ABSTRACT

Cluster managers are among the key software components in compute clusters. They provide an interface to administrators to manage compute nodes, their resources and software, and allow real-time monitoring of both physical and logical states. Workload managers on the other hand provide an interface to the users by allowing allocating available resources for user-defined compute jobs. The goal is to allow users to somewhat abstract their computations from physical nodes by defining jobs on a single "head" node, which, once scheduled, will be distributed among requested resources. Herein we present an open source cluster and workload manager written in Python with minimal dependencies, set up to use containerization as a means to create virtual nodes within the same physical node. Among its applications are partitioning physical resources within one physical node (CPU cores, memory, GPUs) into virtual nodes mimicking a compute cluster, and creating an easy and lightweight solution for running and testing parallel/distributed jobs on minimal hardware.

1 INTRODUCTION

Compute clusters are often comprised of a set of networked computers, each with certain computational capabilities and resources, and they are set up in such a way that multiple users can request said resources for specific experiments or jobs. This is a distributed environment, where compute nodes collaborate to create a single unified system through which researchers, engineers, and scientists conduct computationally heavy experiments based on their requirements, and without having to consider per-node physical hardware requirements. An example of that is in machine learning research, where experiments, depending on their nature and size, require different levels of resources, particularly hardware accelerators, such as general-purpose graphical processing units (GPGPUs) and tensor processing units (TPUs). A standard practice in such scenarios is a cluster management software that is installed on all of the networked computers, or *compute nodes*, and one or more computers manage, organize, and control the rest via network communications made through the cluster manager, which are referred to as *head nodes*. Cluster management software relies on a workload management service or module, which provides an interface for users in order to allow them to request resources, and schedule their jobs accordingly.

SLURM (Simple Linux Utility for Resource Management) [2] is one such service, and is among the most widely adopted among large-scale supercomputers and compute clusters. While the cluster management software keeps track of all nodes, their health status, and so on, it also passes their IP addresses to the workload manager. The workload manager, in this case SLURM, runs daemons, which are processes that start automatically after the operating system boots and continue to run in background, on head and compute nodes. On compute nodes, the daemon is responsible for reporting node resource availability, running jobs, and other relevant information to the head node. On the head node, controller daemons communicate with communicate with compute nodes, which are exposed via their local IP address.

Users then log into head nodes, and use the SLURM interface (commands such as 'scontrol', 'sinfo', 'srun', 'sbatch') to check available nodes and resources, their running jobs, and submit new jobs. The interface is responsible for communicating with the daemon to schedule jobs and report information.

2 BACKGROUND

3 METHODOLOGY

In this section, we describe the proposed approach towards building a minimal-dependency cluster management (CM) service that would operate using the infrastructure provided by Docker [1]. We divide the service into two components: infrastructure and CM software, which are described in the following subsections. An overview of the proposed system is presented in Fig. 1.

3.1 Infrastructure

As mentioned, the service relies on Docker [1] for both infrastructure and containerization as a means to mimic individual physical nodes. This is achieved through creating container images for head and compute nodes separately. While images for both package the software, which will be described in the next subsection, they can differ slightly in terms of dependencies. For example, routines that are implemented only to be executed from the head node need not be packaged into the software on compute nodes, and drivers required to access or utilize computational resources on compute nodes should not be part of the head node image.
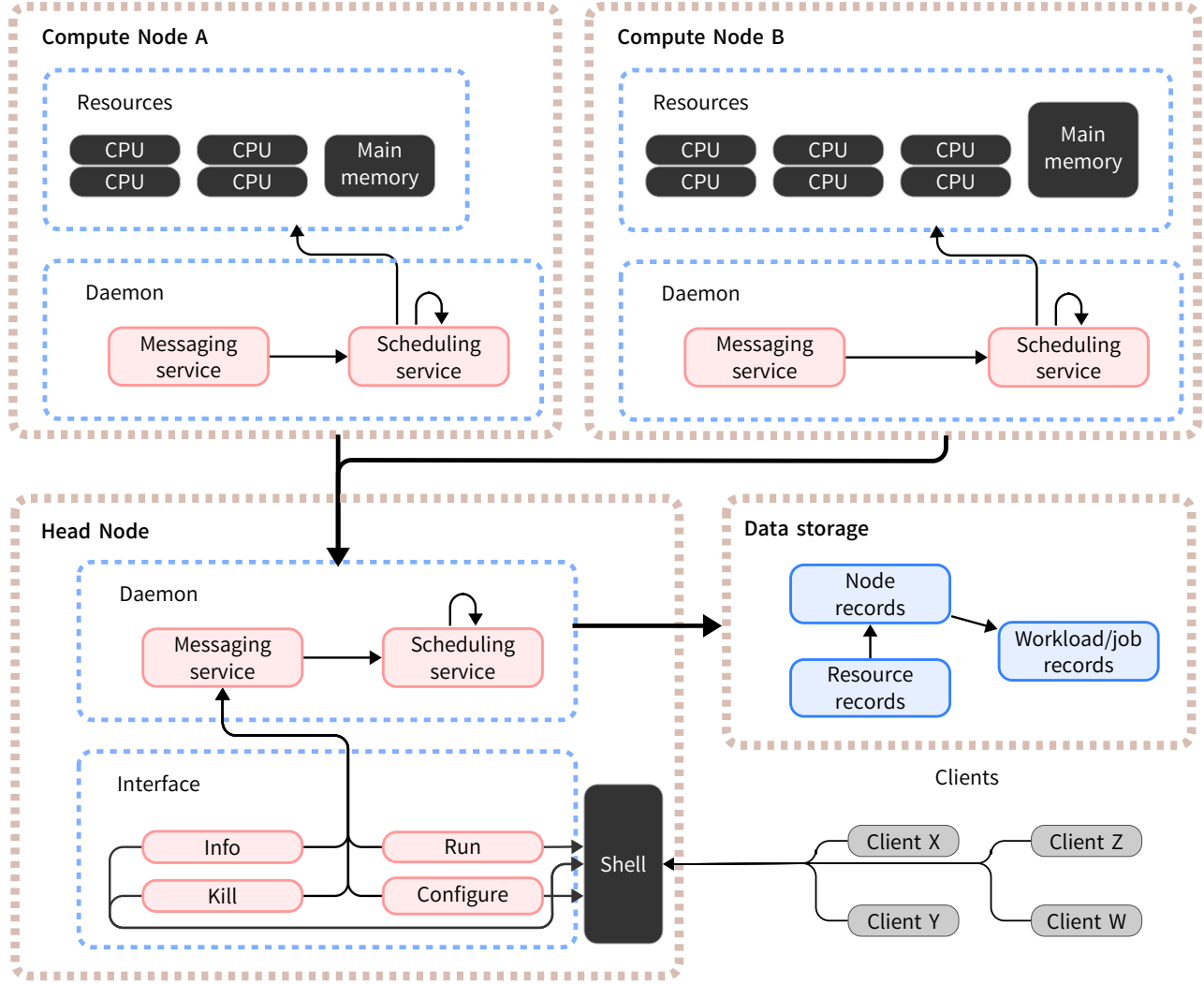
Figure 1: An overview of the components in the proposed cluster manager. Each node will run the CM daemon, which consists of a messaging service, and scheduling service. Upon initialization, the daemon is responsible for detecting available resources, starting the services, and registering itself with the head node. Head node, once itself initialized, will store information on nodes and resources in a local storage. Clients can then use the available interfaces to see available resources, nodes, create/stop jobs, and configure the CM (depending on their privileges.)

Container images used in this work are based on the official Ubuntu image from Docker hub, and include the additional Python 3.10, along with dependencies that are Python packages. The primary dependencies are: Zero-MQ (ZMQ), PyYAML, psutil, and rich. ZMQ socket interfaces are used to implement messaging services used in both head and compute nodes, PyYAML handles parsing the main configuration file that holds the CM default values and settings, psutil provides a Python interface over the operating system's "process status" tool, which is used to monitor the state of running jobs. rich is a command line pretty-printer

used in CM interfaces in order to print available resources and jobs in tables.

In addition to head node and compute node containers, we also containerize the database. Head node keeps records of resources, nodes, and jobs in a MongoDB database, which is implemented in a separate container.

Finally, all services are connected through a docker network between the head node and compute nodes, and a separate network connecting the head node and database element. In addition, we set up mounts to a persistent shared

home directory between head node and compute nodes, following standard practice in compute clusters.

The aforementioned container services are organized using `docker-compose`. Services are exposed to clients through accessing an interactive terminal into the head node container, and compute nodes are simulated by scaling compute node containers.

3.2 Software components

As previously stated, the shared software is primarily written in Python or using Python interfaces. Herein we describe the different components present in the software, starting with communication.

3.2.1 Communication. For the purposes of this project, we write two socket classes on top of ZMQ's request and reply sockets. The reason for that is to provide abstractions to other software components for communication, and reduce code replication, as these sockets will be the basis for communication in the entire system.

By default, the sockets are set up to serialize and transmit pickled python objects, and all communication is assumed to be **synchronous** with **fail-stop** semantics, meaning requests are time-bounded and timed out requests are handled as if the destination is offline.

3.2.2 Abstractions and structures. For ease of modification, extension, and reduced code replication, we introduce the following abstractions and structures into the project:

Job: an object that holds information about a client-initiated process/job. Jobs hold a unique identifier, which we define to be a non-negative integer that is incremented per job submission, but this identifier is assigned to the job once it is accepted by the service. Jobs also hold an optional name, the UID of the user submitting it (so that it may be executed by the same user across nodes instead of as root), the command (i.e. `python compute.py`), working directory and environment variables, optional time limit, and resources requested. Resources requested consist of number of nodes, number of processes per node, number of CPU cores reserved per node, and amount of memory reserved per node. Job instances also hold a status property, which can take values such as `Pending`, `Started`, `Running`, `Completed`, `Killed`, or `Unknown`.

Resource: an object containing a type (CPU, memory, GPU, and the like), a capacity (CPU cores, amount of memory), and a key-value store for allocations. The key-value store matches job identifiers with the amount of the resource reserved (number of cores, amount of memory). Resources therefore implement methods for checking allocations, confirming allocations, and removing allocations.

Node: an object containing a unique identifier (hostname), type (head or compute), and resources. Upon initialization of any service, every node is able to identify itself based on environment variables. This is achieved through Docker's ability to pre-define environment variables for every container. Resources should be identified through either drivers, or a pre-defined configuration, but for the purposes of this work, we define all compute node containers as having 2 CPUs, each with 1 core, and 1 main memory of size 512 megabytes.

Nodes implement multiple methods, some of which are designed to be called both locally and through a remote procedure call (RPC). Some of those methods are described as follows:

- Status check: report whether node is idle or in use. If RPC calls fail, node is assumed to be down.
- Resource allocation: reserve some of node's resources by calling the resource allocation methods.
- Job start: queue and start a job process
- Job end: kill or stop process if still running, free resources allocated for it.

The reason for this formulation is that Node instances can be created in different routines throughout the software, or even fetched from the database. Instead of writing new procedures for finding a node's address and port, starting a socket, and sending a request for an action, all of this can be done through simply calling the methods on the node. If the method is being called on the same node, the local procedure will be called, and if called from a remote node, RPC will be initiated.

Queue: As described in Node, jobs are queued and then started. Queue is an object that holds key-value mappings between job identifiers and their corresponding Job instance, which holds the information necessary to start the jobs (command, working directory, user id, environment variables), as well as process ids (PIDs) corresponding to jobs that have already started for monitoring.

Queue instances run on compute nodes only, and are shared among subprocesses, which is why it is implemented with methods that save its context to file given any one instantiation. This is necessary simply because the main service daemon is broken into different subprocesses, as all communication in this system is synchronous. This will be further clarified in Sec. 3.2.3.

Message: all communication done in this system is done through sending instances of this object. Message instances include the originating node's unique identifier, an action (see Appendix C), an optional response, and an optional content field for transmitting structured or unstructured data.

To further ease communication, RPC calls, and reduce code duplication, most message sending is factored into a single method. It starts by taking in two arguments: node and message. A socket is started at a random port, and then binded to the node's address and receiving port, and message transmission is attempted, and the result is returned. There is an additional routine for sending messages to the head node, which solely relies on the existing configuration and only takes the message as input. This is heavily used in RPC calls and interfaces.

3.2.3 Daemons. Herein we describe the properties of the daemons, which are the central component of the entire system. Both head nodes and compute nodes start off by launching the daemon, and their aliveness is defined by the daemon functioning as expected. Daemons start off with the same procedure on both types of nodes, which is reading environment variables and the configuration file. As described previously, the configuration file holds presets, defaults, and settings, all of which are described in detail in **??**.

Once the daemon identifies itself, meaning it has constructed its own Node instance, it can start communicating. It then launches procedures according to its type (head or compute).

Head daemon: Head node depends on the data storage element, therefore it starts by initializing a connection to the database, which in this case is only initializing the MongoDB connection string. Once completed, the daemon starts its two subprocesses: **messaging service**, and **scheduling service**. The messaging service simply starts a receiving socket at a fixed receiving port (defined in the configuration, and fixed across nodes of the same type), and blocks on receiving messages. Message handling is done according to the action specified in the Message instance (see Appendix C). The scheduling service on the other hand repeats its procedure in small time intervals (defined in the configuration). The procedure is comprised of fetching pending jobs, and calling the workload manager on them in order to attempt scheduling jobs, and if successful, communicating the job to the assigned nodes. The workload manager is described in Sec. 3.2.4.

Compute daemon: Unlike head nodes, compute nodes do not need to connect to the database, since the head node is the only entity in the entire system with direct access to the database. This helps maintain our assumption of a single coordinator, and therefore avoiding race conditions. Once a compute daemon is started, and identifies itself, it first communicates with the head node to "register" itself. Registration is simply comprised of announcing its existence and resources to the head node. Nodes simply transmit a Message with the appropriate action, along with their Node

instance, which includes their identifier, as well as available resources. Upon receipt, head node attempts to create a new node record in the database, or update an existing one with the same node id (in case of a failure and reboot of a compute node.) Once registered successfully, the compute daemon also breaks into two **messaging** and **scheduling** subprocesses, similar to the head node. However, there's a few differences in both. Messaging service on compute nodes, while similar in functionality to the head node, handle a different set of actions compared to the head node, and therefore have their own set of action to handler mappings. Scheduling service on the other hand is almost entirely different from that of the head node. It is comprised of two separate subprocesses, one responsible for starting jobs, and another responsible for monitoring jobs.

The former repeats its procedure in small time intervals, similar to the head node instance in that regard, and checks its local Queue instance for pending jobs. If any are found, it attempts to start the jobs as described by their instance, and logs their process identifiers (PIDs.) Once all replicas of a single job have started and their PIDs are logged, the process reports the job back to the head node, indicating that the job has started running. This triggers an update in the database from the head node, and once all nodes assigned to the job have made the same report, the head node changes the job status to "running" in the database.

The second scheduling subprocess, which is responsible for job monitoring, repeats its procedure is time intervals larger than the previous, leading to its procedures called less frequently (intervals are configurable). Its procedure checks the local Queue instance for *running* jobs, those with PIDs, and simply monitors the process state and runtime. We note that runtime can be both checked both from the process state and also logged by the scheduler when it starts the job in the first place. We choose the latter and therefore avoid resolving possible differences between timestamps on different PIDs corresponding to the same job. Once each job's state and runtime is logged, the monitoring procedure attempts to report jobs that are no longer running (completed, killed, or other similar states) back to the head node. It also attempts to terminate jobs that have exceeded the time limit, if there is one. Head nodes on the other hand again keep track of nodes reporting job completion, and once all assigned nodes report back, triggers a job status update.

3.2.4 Workload manager. We implement a simple greedy scheduler in this work, but in theory any scheduler would be applicable to this system, and can simply replace the existing one. The scheduler is designed to maintain a FIFO (first in first out) ordered queue, which is guaranteed by the ordering in the database according to job ids. Given the ordered set of pending jobs, and an updated the list of available nodes and

their available resources through RPC, workload manager assigns resources to the first job whose requirements can be met. In other words, given any job, the workload manager either finds the specified resource requirements and starts the job, or fails to schedule the job, which means the job will be put back into the queue and checked again later. The procedure also includes communicating with each node separately once to check resource availability, and once to reserve resources and start the job. Because the system is built under the assumption that there is a single head node, or more generally, single coordinator, race conditions are ruled out as a possibility, meaning while the scheduler is attempting to reserve node x's resources for job i, no job j can possibly reserve and take node x's resources before job i gets a chance. A flowchart of the steps taken in the workload manager is presented in Fig. 2.
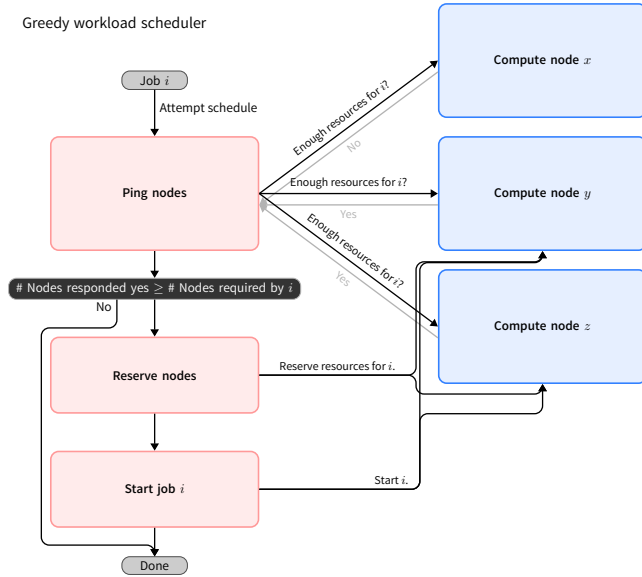

Figure 2: A flowchart of the workload manager.

3.2.5 Interfaces. Interfaces are programs that would be exposed to clients for interacting with the system. Similar to SLURM [2], we implemented separate programs for starting and stopping jobs, as well as checking the overall status of the system. Those programs and their arguments are described below:

cminfo: Prints an overview of the cluster, broken into head nodes, compute nodes, and "valid" jobs (either running or pending). Takes no arguments. A sample output from this tool is presented in Fig. 3.

cmrun: Allows users to set up jobs. Arguments are the following:

- Command is the only required argument. It is the command or commands that will be executed on compute nodes.
- `-a --alias`: an optional job name. Defaults to the first word in command.
- `-n --nodes`: number of nodes to replicate the job on. Defaults to 1.
- `-p --processes`: number of replica processes per node. Defaults to 1.
- `--num-cpus-per-node`: number of CPU cores to reserve per node. Defaults to 1.
- `--mem-per-node`: amount of memory to reserve per node. Defaults to 100.
- `-t --time-limit`: Job time limit in hours. Defaults to 0 (non-positive values represent no time limit).

cmkill: Allows users to terminate / stop their jobs. Only takes 1 argument: job id.

cmconfigure: Allows configuring elements in the cluster, such as nodes, their resources, users, and the like. This tool was not implemented due to certain limitations imposed by the operating system when creating a shared user management service.

Head nodes

Node	Status	# CPUs	Memory	# Free CPUs	Free Memory
ddf755c958ba	Up	2	512	2	512

Compute nodes

Node	Status	# CPUs	Memory	# Free CPUs	Free Memory
11541c0bf7a9	InUse	2	512	0	384
3c0636c23612	InUse	2	512	0	384
3b279dde2676	InUse	2	512	0	256
376c77a1b192	InUse	2	512	0	256

Jobs

ID	Name	User	Status	Command	Nodes
1	scrpe	louie	Running	./mini_scraper.sh -c	11541c0bf7a9, 3c0636c23612
2	minig	larry	Running	./mini_gpt_dist.sh 2	3b279dde2676, 376c77a1b192

Figure 3: A sample output from the cminfo program in an environment with 4 compute nodes, and 2 running jobs.

4 SUMMARY

In summary, in this work we introduce a minimal cluster and workload management system comprised of software components primarily written in Python, and the containerized infrastructure provided by Docker. Containers, container images, container volumes, and container networks help simulate a cluster of nodes, their operating systems, their shared data storage systems, and their underlying network,

respectively. The key software component is the daemon, which identifies nodes, their resources and properties, and sets up communication protocols and scheduling services. Daemons on compute nodes attempt to contact the head node daemon, which serves as the main coordinator. Users primarily use interfaces that are deployed to the head node, which convert user requests to messages that flow through the entire system by first reaching the head daemon, and then possibly inciting messages to compute nodes and back.

5 CONCLUSION

In this work, we introduced a minimal cluster and workload management system, written primarily in Python with minimal dependencies, which can be easily set up and deployed using Docker, in order to create a distributed environment for computational workloads.

While operational, there are a few missing pieces from the implementation, including identification of resources through the operating system and drivers instead of preset values, enforcement of resource limits on user-created jobs, user management, and the ability to scale to multiple head nodes.

In addition, error handling and fault tolerance remains limited. Due to using fail-stop semantics only, and the number of errors that are not tolerated, and as a result are only recoverable with restarting of services, this implementation is not stable enough for standard usage, and requires improvements in that regard.

REFERENCES

[1] Dirk Merkel. 2014. Docker: lightweight linux containers for consistent development and deployment. *Linux journal* 2014, 239 (2014), 2.

[2] Andy B Yoo, Morris A Jette, and Mark Grondona. 2003. Slurm: Simple linux utility for resource management. In *Job Scheduling Strategies for Parallel Processing: 9th International Workshop, JSSPP 2003, Seattle, WA, USA, June 24, 2003. Revised Paper 9.* Springer, 44–60.

A   SOURCE CODE

The entire implementation is available in the following GitHub repository, under the `project/` directory. https://github.com/alihassanijr/CS630-DistributedSystems.

The only requirement for starting the service is Docker, docker-compose, GNU make. While in theory there should be no issues with running this service on Windows, it should be noted that it was only tested on Unix-based operating systems.

B   CONFIGURATION

C   ACTIONS