Cluster Middleware

Divyang Mittal Siddhant Agarwal Shrey Shrivastava

Ritik Kumar Kanishk Singh

Abstract

The need for more and more computation has grown exponentially over the years. It was only 10 years ago when deep learning era started and now we have models that contain more than 150 billion parameters. Every year we see better supercomputers being developed in terms of the computation power. We all are aware that the Moore's Law was saturated that means the development in cpus has slowed down. So, how exactly are the increasing demands for computation met? The answer is to combine several cpus to distribute the workload among them. Clusters are networks of several computers that can be viewed as a large very powerful computer. Each computer (called a node) is itself (ideally) a high performance computer with a powerful cpu. Resources are allocated from these computers for the jobs submitted by the user. Clusters have become an integral part of any high computing facilities. Several modern supercomputers are infact huge clusters of compute nodes. A cluster management service or a cluster middleware is a system that manages this network of computers. It performs several necessary functions such as job scheduling, fault tolerance, load balancing etc. We discuss several possible implementation paradigms of these services. We also provide a simple prototype for the cluster management system that will perform most of the necessary functions.

1 Introduction

With the ever increasing scale of deploying applications, the need of managing hundreds of thousands of machines to run them efficiently with high availability has led us to develop complex system software. Other computing applications which require machines to have processor and memory capabilities impossible for a single machine, require a group of machines to act as a single machine. A cluster is a group of machines, usually heterogeneous in terms of processor, RAM, disk, and I/O capabilities linked through a high speed LAN, acting as a single logical unit i.e a single more powerful machine. Almost all research groups that require high performance computation need clusters. Although small research groups have clusters with as few as 8-16 compute nodes.

The power of computer clusters can be judged from the example of supercomputers. Few decades ago, supercomputers such as Cray-1, Cyber 205 were computers with capabilities such as vector processing. With time all these enhancements have been incorporated into personal computers. Modern supercomputers are hence made up of large number of such processors. According to top500[5], the list of fastest supercomputers of November 2020 is dominated by computer clusters such as Fugaku, Summit, etc. Fugaku[8] is the fastest supercomputer according to that list and consists of 158,976 computation nodes. The second entry in the list is Summit[10] by IBM systems which runs over 4,608 nodes. In fact, Most of the supercomputers are cluster systems. These consists of several thousands of high performance compute nodes. Cluster computing, at this scale, is being used by many

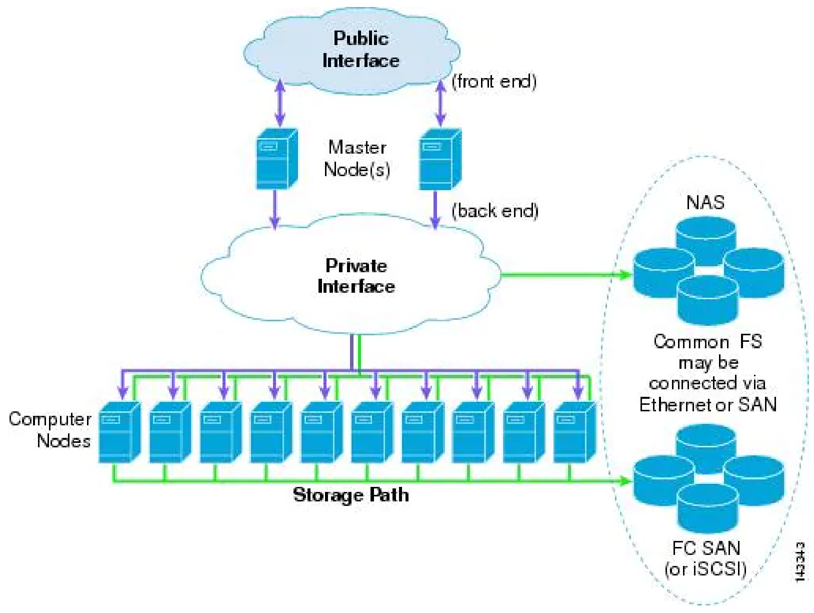

Figure 1: A sample architecture for a cluster system. The user interacts with the master node which communicates with the compute nodes. The user, in general, cannot access the compute nodes directly. Courtsey: [1]

product based companies and scientific research organisations. The jobs performed by cluster computers usually involve executing 10000 copies of product applications or a single application with 100 GB RAM requirement.

Cluster computing provides a single system image (SSI) of the cluster so that users can access the resources without worrying for the underlying complexity of combining the cluster resources, achieved using cluster middleware. Cluster middleware creates an abstraction for the user, by gluing the operating systems of the cluster nodes and providing a unified access to resources.

Cluster middleware acts as an interface between user applications and cluster node resources. It is composed of resource management, job queuing, job scheduling, job management, and fault tolerance.

2 Design Issues and Architectures

A cluster management system consists of a central node that performs various management protocols such as job scheduling, matchmaking and load balancing. There are several design principles for each of the protocols, that can be followed while designing a cluster middleware.

An example can be deciding whether compute nodes can be accessed by the user or not. Several clusters do not allow the user to directly connect to the compute nodes. Here, the user can simply connect to the master node and the master node then schedules jobs on the compute nodes. But, systems like Condor [14] allow jobs to be directly initiated at the compute nodes. In Condor, the user system becomes a part of the cluster, starts the job and sends the status to the master node. If jobs are submitted to the master node, they are scheduled on the idle user systems which act as compute nodes.

Network topology is another such design choice. In most cases, the compute nodes are directly connected to central server. They may or may not be in a fully connected topology among themselves. Beawulf clusters are perhaps the most simplest clusters. These have a dense topologies with a master-slave architecture.

We now discuss the various design policies for cluster middleware.

2.1 Job Scheduling

Job scheduling is about finding the most eligible job to run from various jobs available for execution. It is often broken into various steps. Now, We explain some of these important processes involved in job scheduling.

2.1.1 Matchmaking

The first step decides the node at which a job should be accepted for execution. There might be a central master node to handle this or can be handled in a distributed way at each node. The jobs might have some priorities and different resource requirements. Therefore, scheduling the job and finding the best node for the job based on priorities and requirements is important. Matchmaking is essential for all these requirements.

The most commonly implemented architectures requires the jobs to be submitted to the central server which schedules these on the compute nodes. Even for non dedicated cluster or workstations, there exists a central service which has maintains the status of all available nodes in the system.

Every job mentions some system requirements like memory, number of processing units etc. Jobs can also be allowed to run on more than one compute nodes. Also, different jobs can have different priorities. First of all the central manager decides on the eligible machines for each job depending on the basic requirements. It also decides on how much resources can be allocated to each job depending on their priorities.

2.1.2 Load Balancing

One of the most important tasks for the central server is load balancing. Some architectures follow non preemptive setup where once a job is dispatched, it cannot be disturbed. But some paradigms allow the central server to preempt jobs. This is sometimes a desired quality and allows the system administrator to forcefully stop the misbehaved jobs. The server can also be allowed to migrate jobs from one node to the other to ensure load balancing. The server can also be allowed to follow dynamic load balancing where it will constantly monitor the states of the nodes and migrate jobs if necessary, but this will lead to a lot of overhead on the server end.

2.1.3 Fairness Policies

The servers maintain priority queues for the various jobs while scheduling jobs. This happens both at central server and at computation nodes. But the server should ensure that a job does not constantly starve by dynamically changing the priority of waiting jobs. To prevent this age of job is considered while scheduling them. But if a job with high priority arrives, some systems may preempt a running job to dispatch the high priority one.

Although, we have been constantly saying that the central node dispatches jobs to the compute nodes, this assumes that the central node simply manages the cluster. But some systems allow the central server to act as a compute node as well. There are systems where the user can submit jobs to any machine in the cluster and this machine informs the central node about the job it is running. The central node maintains the state of the other nodes and follows simple load balancing primitives. Clearly this system should follow a completely connected topology.

2.1.4 Parameters and Metrics

Different parameters are used to decide the order in which these jobs should be executed. Some of them are a pre-specified job priority(niceness), deadline associated with the job, estimated execution time of the job, availability of computing resources, dependencies of jobs, nature and size of job, capability of compute nodes in case of heterogeneous system and job submission time.

The various metrics used to judge the performance of a job scheduler in this context are fairness in allocation of resources, maximum utilization of resources, maximum throughput and minimum response time. A poor cluster management system design might starve low priority jobs to prefer jobs with high priority. Thus, it is equally important to consider some sort of fairness to ensure the low priority jobs do not get repeatedly preempted or in other words, ensure no starvation. One major goal of job scheduling is to achieve load balancing over various nodes. A good job scheduler ensures proper load balancing by distributing equivalent number of jobs to these nodes.

2.2 Fault tolerance

Faults are extremely common in any distributed system. These faults can arise due to both hardware or software failures. As the system complexity increases, the number of possible fault points also increases. A good distributed system will try to handle the faults such that the user never finds any occurrence of faults. But this is not always guaranteed. Moreover, every fault tolerant system have a maximum number of faults that it can handle. It is not possible to create a system that can handle any number of faults.

To handle faults, redundancies need to be added to the system which increases the cost of the system. So there is a trade off on the number of faults the system can handle. Fault tolerance can be achieved in several ways and is dependent on what type of faults are being handled. For example if the disk crashed, then the data stored in the disk is lost. To handle this, the system must maintain at least one copy of the data in some other disk. Also the replication should be kept updated otherwise a lot of transactions performed on the data can be lost. But then how often should the data be updated? Also, several disks are collectively stored in stacks that are interconnected. What if an entire stack fails with the replicated data being on the same stack as the original data? There are several protocols that handle the replication of data.

If say any compute node crashes, then the job that was running in the compute node will be lost. So, the job must be restarted at some other compute node. But then should the job be started from the beginning? If so, the data has already been updated while the job was running, how to get back the disk state at the beginning of the job? Also restarting the job from the beginning can lead to a lot of wastage of computation and time.

Clearly, fault tolerance requires several design principles to be followed and solving all kinds of faults is very expensive for the system. So the designer needs to decide on type and number of faults the system will handle.

A simple way to handle the crash of a compute node can be periodically save the checkpoints of the nodes and jobs, and restore them once the fault is removed. This can be followed if the time taken to recover is small. But if the time to recover is large, then the jobs in the node will be stalled for a lot of time. In typical systems such as Condor, a central manager is present which periodically sends heartbeats to all nodes. This checks for any potential crash failures at any nodes who reply to show their presence. In case no reply is received within a fixed time interval, the node is assumed to have crashed and is removed from the system. All the jobs scheduled at that node need to be re-deployed at a new node. Here, if the jobs at that node are restarted, then it must be ensured that the files and data can be restored to the initial state. If the jobs can be restored from checkpoints, appropriate checkpoints for the relevant files must also be maintained.

If the central node or the master node fails, the situation is different. There can be more than one master node, where one can be active while the other can kept as backup. There can be more than one active master nodes as well, but then there must exist a mechanism for the master nodes to inform each other if a job is submitted. If the compute nodes can run the daemons of the master node, (like Condor), a new master can be elected from existing nodes via leader election protocol. The central server can periodically save it's global snapshot on some backup server so that this new central manager can restore it's state.

2.2.1 Dependability

The system designer must ensure that the cluster is dependable. This means that the user can depend on the system and can assume that in most cases, he will be able to successfully execute the jobs on the cluster. But dependability is an abstract term. There must a metric to compare two systems. Also, the designer must keep in mind the time taken for the fault to be repaired. If the time to recover from a fault is small, then the system can assume small number of simultaneous faults. But if the recovery takes days, then the designer must design the system accordingly.

Mean Time To Failure (MTTF) measures the average time a device can run before failing. Mean Time To Repair (MTTR) is the average time taken to repair the device that had failed. Uptime is the percentage of time the device was working and was available. These metrics can help to compare different designs for their fault tolerance capabilities. If a system consists of several devices with low MTTF, and high MTTR, it means that there will be lot of components that might not be available simultaneously.

2.3 File system management

File system is another major component of a distributed system architecture. The data needs to be share with the computation nodes and movement of data is required between the nodes as well. Due to this, There is a large amount of data in the cluster moving from one node to another. This requires an efficient file system which can help us store the data in an organized manner. There are two options available to us with respect to file systems, local or centralised file system. Systems with only local file systems have a quicker computation when they have data stored with them but a centralised file system ensures swifter communication between nodes. In case of crash, data stored in local file systems is lost but with centralised file system this data is secure.

Several cluster management systems use a separate file server. Here, the compute nodes communicate with "centralised" server to access the required files for execution. The centralised data server can itself be a distributed system, implementing a Network Attached storage protocol. So, the compute nodes still have a centralised view for the files but these files are distributed over a network. Moreover, the file system can be made a part of the cluster with the centralised service only providing the block level information. Naturally, all compute nodes should be directly connected to the file server.

Cluster services also allow the compute nodes to store files and itself form a distributed file system like NFS. But this has some limitations. First of all, this will call for the compute nodes to be completely connected with each other. Also, compute nodes should ideally be performing computation intensive tasks. Running a file system over them could unnecessarily waste compute resources. There can be caches at each nodes in accordance with the principle of locality to improve performance. The performance of a such a file system can be judged by testing some data intensive programs on this cluster.

2.4 Administrative policies

Generally, there are administrators who monitor clusters and make sure that the users face no issues with the cluster middleware. Naturally the cluster middleware should provide additional privileges to the administrator. Cluster managers like SLURM allow the administrator to design the configuration of the cluster which includes the management policies, scheduling policies etc.

So, the administrator must be allowed to monitor the state of the nodes, and view the status of all jobs and job queues. Moreover, the administrator should also be able to kill any job. Each user (user group b extension) needs to have an upper limit on the resources used, and running time. It should not happen that one user, or a group of users, are continuously using all the resources, while other jobs are constantly waiting. But, some users may be allowed a higher priority in running jobs. These decisions are made by the administrator. The admin defines the **quota** and priority for each user or user group.

The cluster middleware designer must make sure to include some admin privileges. It is up to the designer to allow the admin to be able to preempt jobs or allow to define the priorities of jobs etc. Some cluster manager give a lot of authority to the administrator, while in some, the administrator is just a person who monitors the nodes and accounting records.

3 Case Study - SLURM

3.1 Introduction

Slurm [19] is an open source, fault tolerant and scalable cluster middleware for linux clusters. It was initially developed as a free software resource manager in 2002. It was then known as Simple Linux Utility for Resource Management. Later in 2008, several sophisticated plugins were added in 2008 and it slowly became one of the most popular cluster middleware used by both small and large clusters. Even several supercomputers use the Slurm Workload Manager to manage its resources and schedule jobs. One example can be the CTE-POWER supercomputer in Barcelona. Even the supercomputer in our institute, PARAM Shakti uses Slurm.

Slurm primarily performs three functions as a cluster manager,

- It allocates exclusive and/or non exclusive access to computation resources for some duration.
- Provides a framework to execute and monitor jobs on a set of allocated nodes.
- Maintains queue of pending jobs to and allocates resources to the jobs in the order they are present in the queue.

We will discuss some design goals that were discussed by the Slurm developers. Then we will dive deep into the architecture, where we will explain the different daemon processes that perform the various operations. Later we will discuss the interface that is provided by slurm to the user to submit and manage jobs. Finally we will explain the advantages and disadvantages of the various design choices made by slurm.

3.2 Design Goals

SLURM has been developed with a view to provide a resource management system with a design that is simple, efficient, fault tolerant, highly scalable and portable. This project had the following design goals as listed below -

- SLURM has maintained a simple architecture which provides an easy first system to users to understand cluster management system. Also it is an open source project making it freely accessible for everyone.
- SLURM is highly scalable which allows it to add thousands of nodes while maintaining it's simple design.
- SLURM has a plug-in mechanism making it easily portable for various systems. Initially it was created for Linux but support for various other UNIX OS can be easily added.
- SLURM uses a similar plug-in design for the interconnects and supports UDP/IP based communication.
- SLURM ensures security by authenticating users and communication between services using crypto techniques.
- SLURM is fault tolerant and aims to provide efficient use of resources. This is done by making a node quickly available for subsequent jobs after completion of a job. The job submitted to a crashed node is allocated to a free node without affecting other nodes.

- SLURM maintains general configuration files which makes it easy to maintain for the administrators. Changes made in the these files can reflect in the system without interfering with the running jobs. It also uses general purpose scripts to maintain it's interface.

3.3 Architecture

Slurm has a central manager that schedules and manages jobs. As shown in figure 2, there is a central daemon slurmctld running in the central manager, while each compute nodes run slurmd. There is a general purpose plugin mechanism that provides different behaviour such scheduling policies, process tracking etc). Slurm may be aware of the network topology and use it for node selection. No user can directly interact with any of the compute nodes. It provides a rich set of command line options to control the selection of nodes and distribution of task to the allocated nodes.

First let us discuss about the various daemons that run on the nodes.

3.3.1 Daemons

Daemons are background processes that run continuously and are primarily for handling service requests. There are four daemons that are particularly important to manage the requests in slurm.

- **slurmctld (Central Daemon)**: It is the central manager daemon that runs on the central node. When it starts, it first reads the config files and then the additional state information stored in separate checkpoint files from previous execution of slurmctld. It monitors the state of each node in the cluster. It periodically queries the slurmd (local daemons) running on the compute nodes to receive state information. It accepts user job requests and places them in appropriate queues. It also allocates resources to the jobs and keeps track of the hot spare pool.

- **slurmdbd**: This is the slurm database daemon. This is used to record the accounting information in a database. It uploads the configuration details like limits, fair-share etc to slurmctld.

- **slurmd (Local daemon)**: This is a small, light-weight, multi-threaded daemon that runs on the each of the compute nodes. It acts like a remote shell, i.e. it waits for work, executes it, returns the status and then again waits for another work. It initiates the jobs and so runs with root privilege. It only has the information of the currently executing jobs. Initating the process includes setting process limits, establishing environment variables etc. It also allows the handling of stdout, stdin and stderr for the jobs. It also provides fault tolerant hierarchical communications with configurable fanout.

3.3.2 Plugins and Configuration

Slurm is very flexible and easily configurable to show different behaviours. The administrator decides on the various policies for scheduling, management etc and puts them in the respective config files. Slurm provides a general purpose plugin mechanism to support the various infrastructures. This permits a wide variety of configurations. Plugins are basically dynamically linked object files that are loaded at run-time based on the configuration files. In the configuration file, the admin lists down all the plugins for the various policies that are to be followed.

Management Plugins: These define the policies for location of controllers, logs, backups, state info etc, authentication policies, policies for checkpointing, accounting, security and encryption, process tracking etc.

Scheduling plugins: These basically contain the policies for the type of scheduler, preemtion and priority. Scheduler type can be the default FIFO scheduler or backfill scheduling policies. In Backfill scheduling, the jobs are scheduled as long as they do not delay any process waiting higher in the

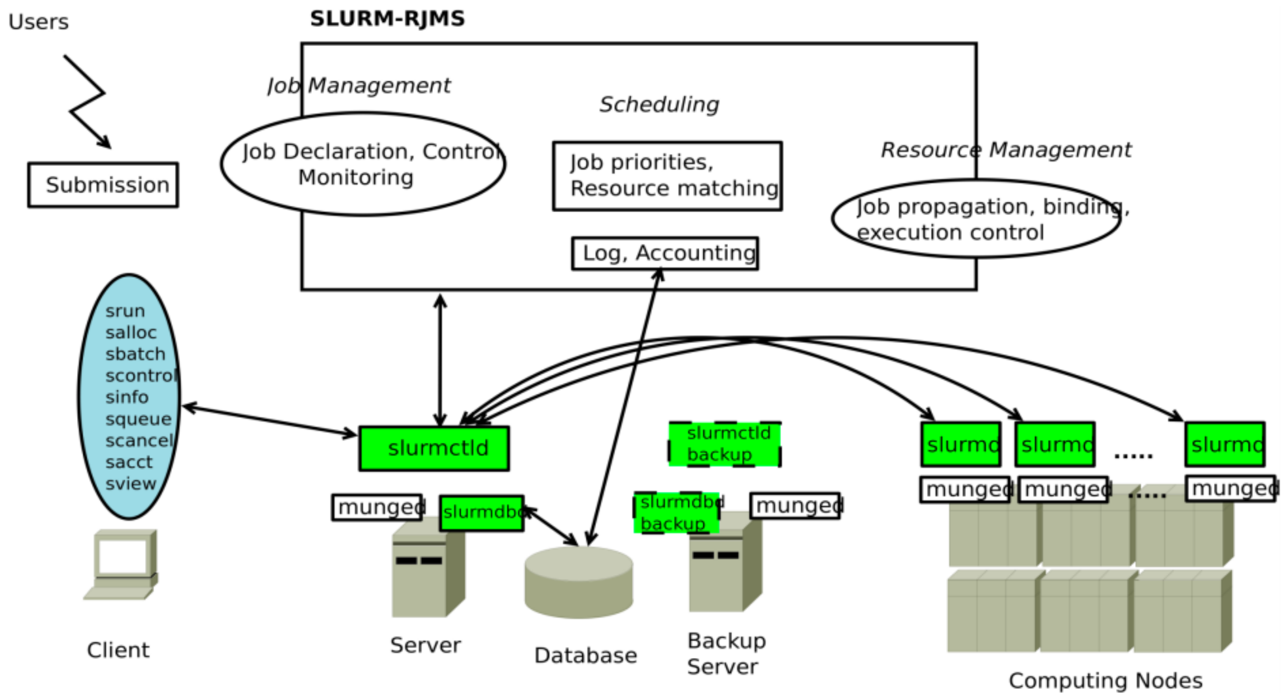

Figure 2: An overview of the slurm architecture. Client(user) is given an interface in the form of command line instructions. slurmctld daemon which runs on the central manager accepts the user requests. slurmd daemons run on the compute nodes and act as remote shells to slurmctld. slurmdbd maintains the database. The figure also depicts a backup central server to be used during faults. Courtesy: [13]

queue. There can be several preemption modes like simply cancelling a job that is preempted or saving the checkpoint. The preempted jobs can also be re queued.

Allocation plugins: There are polices defining how the resources are allocated to the tasks. One approach can be to allocate entire nodes to the tasks, i.e. resources are shared. The other way can be to allocate individual resources to jobs without sharing any resource. There can also be policies for affinity and binding of tasks to cpus.

Partition plugins: Slurm divides nodes into sets called partitions. These partitions are defined in the config file. Each partition has some specific properties like preemption and priority. That means a partition can have lower priority than other partitions. Partitions will be discussed in detail in section 3.3.3.

Topology plugins: If slurm is aware of the topology of the compute nodes, it can provide a best-fit selection for resources. Moreover, each nodes maintain a separate config file that contains the description of the topology architecture.

Accounting plugins: There a list of plugins for configuring the resource accounting. There are plugins to control how the data will be recorded, to gather information from the database, to control how job completion information will be recorded etc.

There are many more plugins apart from what have been mentioned. Infact, there are 100+ plugins in 26 different varieties.

3.3.3 Node architecture

The various nodes in the system are grouped into disjoint sets known as partitions. A job thus requires an allocation of a partition for some amount of time. Each of these partitions have their own specifications such as size of job, time limit, etc. A

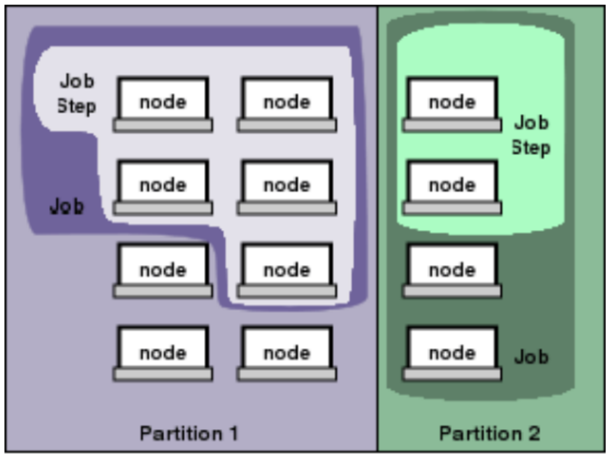

Figure 3: Depiction of partitions in slurm. Here, various computation nodes are divided into two partitions. Courtesy: [15]

partition may have multiple jobs as long as it has sufficient resources to run them. A job can only be assigned to a particular partition. After this allocation, these jobs can run independently for the specified time. SLURM also manages the resources between the multiple jobs in a partition to prevent any competition and subsequent starvation.

There are several advantages of using multiple partitions. Different partitions provide different users with different capabilities. Often partitions have different priorities so multiple queues are maintained each with different priorities. Partitions group similar machines in the cluster to form a subset. For example, a person with a gpu intensive job should ideally be waiting only for other gpu-intensive jobs. Also assigning such jobs to a machine designed more for cpu intensive jobs means resources are not being utilised properly.

An interesting use case of this partition system has been used in PARAM Shakti [11]. It has divided the nodes into various partitions depending on their computation power. The various divide include standard, gpu, hm, standard-low and gpu-low. These also differ in the availability of gpu's as evident from their namesake. This is beneficial as different amount can be charged for providing these varying level of computation power.

3.3.4 Fault Tolerance

SLURM employs various methods to ensure that this system is fault tolerant. The central daemon, slurmctld runs over two nodes, one in master other in standby mode, to ensure the system can recover in case the master node crashes. This master node periodically writes checkpoints to the disk. The recovery process involves the backup central to read the configuration file. Then it restores the checkpoint file for recovery of previous state.

SLURM provides a configuration nonstop.conf [15] if we want to run it in fault tolerant mode. It groups the nodes in two additional sets namely failing nodes and failed nodes. Failing nodes are those which are malfunctioning or are expected to fail. There are also a cluster wide set of nodes known as hot spare pool. These are responsible for handling the jobs with the failing or failed nodes. Any failed node on recovering gets added to the spare pool by default. For the application, SLURM provides them with a replacement node or increases their run time.

3.4 User Features

3.4.1 Command Line Interface

SLURM provides a simple command line interface to the end user to run their jobs. These all have been bundled into the SLURM APIs as follows-

- **scancel:** Used to cancel a job or send a random signal to all processes performing a particular job. The user must have proper authorization to use it.
- **scontrol:** An administrative command used to remove a node or partition from the cluster.
- **sinfo:** Provides information about the various nodes and partition.
- **squeue:** Shows the queues of running and waiting jobs.
- **srun:** Used to allocate resources and run jobs.

3.4.2 Communications Layer

SLURM provides flexibility in choosing the communication layer due to it's plug-in mechanism. At the moment it uses Berkeley sockets. It is capable of serving around 1000 nodes using ethernet and sockets.

3.4.3 Security

SLURM provides certain privileged instructions such as scancel and scontrol that can be used by authorized people only. Also the SLURM configuration files can only be modified by system administrators. SLURM supports the following authentication mechanism - munged,authd and none. Implementation for other authentication mechanisms can be developed using it's plug-ins.

The various steps of a SLURM job are authenticated using an encrypted job step credential. It contains many details including the user ID, job ID, ist of resources and lifetime of this credential. This is provided to the user upon calling srun. This credential is checked by the slurmd daemon to allow access to resources.

3.4.4 Job Modes

SLURM provides three different modes to user to run a job. These are interactive mode, allocate mode and batch mode. In interactive mode, the stdout,stderr and stdin requests are redirected to the user terminal. A user can provide any inputs or communicate to the job using this terminal. In the batch mode, The jobs are continuously accumulated in a queue as long as resources are available by slurmctld. SLURM submits this job to the slurmd as soon as the resources are available depending on it's priority. This submitted job maybe an srun or multiple instances of srun in a job script .In allocate mode, a user is allocated a job. The user can then run the job manually or by using a script. Upon job allocation, a terminal is spawned for the user and the job is complete when the terminal is closed.

4 Case Study - Linux HA

4.1 Introduction

A high-availability cluster is a group of computers to improve the availability of services and resources that cluster provide, so that a failure of any single node in the cluster will not cause the service to become unavailable. This is done by a cluster software. The software monitors the availability of the cluster nodes and the availability of the services that are managed by the cluster. Anytime, if the HA cluster notices any server going down, it makes sure that services is restarted in any other servers in the cluster, so that can be used again with little interruption.

To provide high availability in Linux [4] , it started with a simple code implementation called Heartbeat [12] . An early implementation of Heartbeat could monitor only two nodes and could start more than one services on those two nodes. So, if a node with some resources/services went down, the resources gets restarted on the other node. Some of these shortcomings were resolved in the second version of Heartbeat. Heartbeat 2 is based on the use of Cluster Resource manager which allowed to configure two or more nodes in active/ passive or active/ active configurations. Later the Heartbeat 2 Cluster Resource manager was removed and a new project called Pacemaker was started. Our current case study is based on the project Pacemaker which is used in most of the High Availability clusters today.

4.2 Pacemaker-(Linux-HA)

Pacemaker [7] is a high-availability cluster resource manager that can coordinate the startup and recovery of interconnected services across a group of machines in order to maintain the integrity of desired services and reduce resource downtime.

A Pacemaker stack is built upon following core components. An overview of the Linux High-Availability(HA) Cluster-Stack is shown in Fig 5:

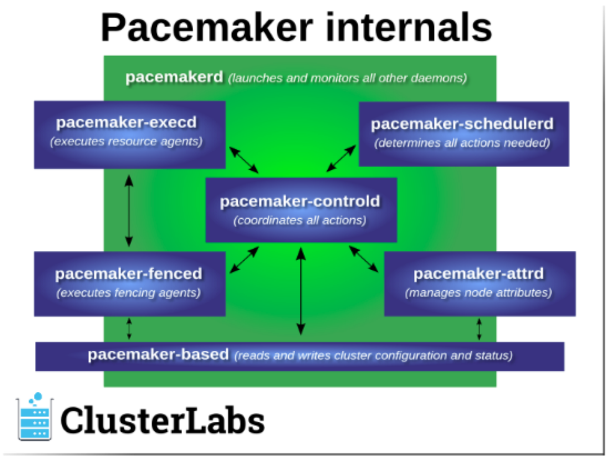
Pacemaker internals
pacemakerd (launches and monitors all other daemons)
pacemaker-execd (executes resource agents)
pacemaker-schedulerd (determines all actions needed)
pacemaker-controld (coordinates all actions)
pacemaker-fenced (executes fencing agents)
pacemaker-attrd (manages node attributes)
pacemaker-based (reads and writes cluster configuration and status)
ClusterLabs

Figure 4: Pacemaker-Architecture: [7]

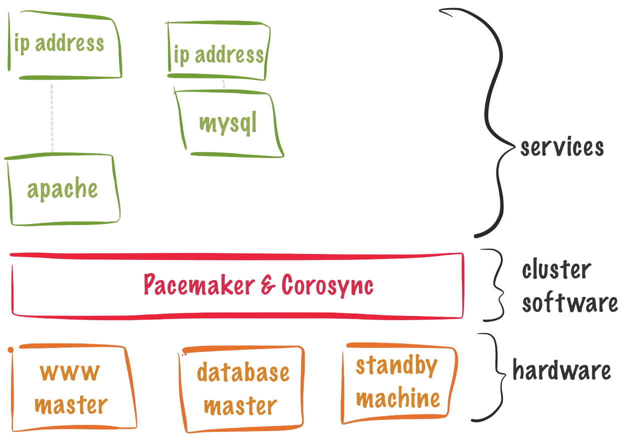

Figure 5: Cluster-Stack: [7]

- **Resource Agents:** The abstraction that allows Pacemaker to manage services is called Resource Agents. They contain the logic for the cluster to execute when it decided to run, stop, or check the service's health.

- **Fencing Agents:** Fence agents are the artificial construct that allows Pacemaker to isolate bad nodes by turning them off or restricting their access to common resources.

- **Cluster membership layer:** This component provides reliable messaging, membership, and quorum information about the cluster. Currently, Pacemaker supports Corosync as this layer. Corosync APIs provide membership (a list of peers), messaging (the ability to communicate with processes upon these peers), and quorum (do we have a majority) functionality.

- **Cluster Resource Manager:** The pacemaker connects to the brain, which processes and reacts to events in the cluster. Nodes entering or abandoning the cluster; resource events caused by failures, maintenance, or scheduled activities; and other administrative decisions are examples of these events. Pacemaker can start and stop resources and fence nodes to reach the desired availability.

- **Cluster tools:** These provide an interface for users to interact with the cluster. Various command-line and graphical (GUI) interfaces are available. In addition to the command line tool and the Pacemaker GUI, the High Availability Extension also comes with the HA Web Konsole (Hawk) [3], a Web-based user interface for management tasks.

4.2.1 Architecture

Pacemaker is composed of multiple daemons that work together. An overview of the Pacemaker architecture is shown in Fig 4 :

- **pacemakerd:** It is the master process that releases all other daemons and even restarts the daemons if they exit unexpectedly.

- **pacemaker-based:** This is the manager that keeps the CIB synchronised across the cluster and manages requests to modify it. The Cluster Information Base (CIB) is an XML representation of the cluster's configuration as well as the state of all nodes and resources in the cluster.

- **pacemaker-attrd:** It is the attribute manager which is in charge of maintaining a database of all node attributes, keeping it synchronised across the cluster, and handling requests to modify them. In most cases, the attributes are recorded in the CIB.

- **pacemaker-schedulerd:** The scheduler decides what actions are required to achieve the cluster's desired state given a snapshot of the CIB as feedback.

- **pacemaker-execd:** It is the local executor that processes and returns requests to run resource agents on the local cluster node.

- **pacemaker-fenced:** The fencer handles requests to fence nodes. Fencer chooses which cluster nodes to execute which fencing devices given a target node, then calls the appropriate fencing agents and returns the result.

- **pacemaker-controld:** The controller is Pacemaker's coordinator, keeping track of cluster membership and orchestrating all other components.

4.3 Cluster Resource Manager

4.3.1 Cluster Resource

A resource is a service made highly available by a cluster. Cluster resources can include web sites, databases, file systems, virtual machines, and any other server-based applications or services one want to make available to users at all times. A resource can be of primitive type, or can be of complex forms such as groups and clones. Each of the primitive resource comes with a resource agent that abstract the service it provides and present a consistent view to the cluster. So the cluster itself does not need to understand and manage resources but depends on the resource agent to do the right thing on start, stop operations on the resource.

Cluster performs various operations/actions on a resource by calling the resource agent. Some of the most common operations which a resource agent must support are start, stop and monitor.

4.3.2 Monitoring Resources

If user wants to make sure a resource is up and running, one should set up resource tracking for it. If resource monitoring configuration is not done, then the cluster will always show the resource as healthy even if the resource is failed after a successful start.

The CRM performs a probe, or initial monitoring, for each resource on each node to ensure that it is running where it should be and not where it shouldn't be. A probe is also executed after the cleanup of a resource. To ensure that resources remain healthy, user can add multiple monitoring operations to a resource's definition. The CRM will choose the one with the shortest interval and use its timeout value as the probe's default timeout. If no monitor operation is configure, the cluster-wide default operation applies.

4.3.3 Resource-Score

The cluster's operation is dependent on scores of all types. Everything from migrating a resource to choosing which resources in a deteriorated cluster to stop is accomplished in some way by manipulating scores. Scores are assigned to each resource, and any node with a negative score for that resource will be unable to use it.

4.3.4 Resource Constraints

Configuring all of the resources is only half of the work. If the cluster is aware of all required resources, this might not be able to control them properly. Users can define which cluster nodes resources can operate on, what order resources will load, and what other resources a resource is reliant on using resource constraints. Three different types of constraints are available:

- **Resource Location:** The cluster uses location constraints to determine which nodes a resource can run on.

- **Resource Colocation:** Colocational constraints simply tell the cluster which resources are allowed or not allowed to run on the same node.

- **Resource Order:** Ordering constraints specify the order in which certain resource actions should take place in the cluster.

4.3.5 Resource Allocation-Policy

As per the resource allocation scores on each node, Pacemaker gets to decide where to assign a resource. The resource will be assigned to the node with the highest score for the resource.

In the event of a tie, Pacemaker will balance the load by selecting the node with the fewest allocated resources. We can't optimally balance the load based on the number of resources allocated to each node. Furthermore, if resources are allocated in such a way that their combined requirements exceed the available capacity, they may fail to start or perform poorly.

To take these factors into account, Pacemaker allows the user to configure:

- **Utilization Attributes:** Utilization attributes in node and resource objects may be used to configure the capability that a node provides or that a resource requires. Users can name utilisation attributes according to their inclinations and define as many name/value pairs as needed.

- **Placement Strategy:** After configuring the capacities provided by nodes and the capacities required by resources, the user must set the placement-strategy in the global cluster options.

4.3.6 Resource Priority and Preemption

Perhaps not all resources can be activated at the same time. In that case the cluster can disable lower-priority resources in order to keep higher-priority resources operational. The tie is broken through the following procedure:

- To avoid resource shuffling, the resource with the top score on the node where it's running is allocated first.

- If the scores above are identical or the resources are unused, the resource with the highest score on the chosen node is assigned first.

- If all of the above scores are identical, the first usable resource mentioned in the CIB is allocated first.

4.3.7 Handling Resource Failure

CRM automatically attempts to recover failed resources by restarting them. If it fails to be achieved or it fails N times on the current node, it will try to run over on another node. User can define the number of failures threshold for resources, after which they will migrate to a new node.

4.4 Corosync

Corosync [2] is the communication layer of modern open-source clusters. It is the cluster membership layer that monitors the availability of nodes. It manages and monitors node membership.

4.4.1 Features

Main features provide by Corosync Project are :-

- A closed process group communication model with virtual synchrony that guarantees to create replicated state machines.

- A quorum system to notify applications when quorum is reached or lost.

- A simple availability manager to restart the application process upon its failure.

- A configuration and statistical in-memory database that allows you to set, recover, and receive information change alerts.

4.4.2 Quorum

If more than half of all possible votes are successfully cast, the cluster is operational. The quorum is the number of votes required to obtain more than half of the votes. If half or more of the nodes in a cluster can't connect with each other, the cluster loses quorum.

4.4.3 Working Principle

Corosync uses the totem protocol for "heartbeat" like monitoring of the other node's health. Each node receives a token, performs some work such as acknowledging old messages or sending new ones, and then passes the token on to the next node. This constantly goes around and around. If a node fails to pass on its token after a short timeout period, the token is declared lost, an error count is increased, and a new token is sent. The node is declared lost/dead if it loses too many tokens in a row.The surviving nodes form a new cluster after the node is declared lost. The new cluster will continue to provide services if there are enough nodes left to form quorum.

4.5 Fencing

Fencing guards against data corruption caused by faulty nodes or unintended concurrent access to shared resources. Fencing prevents a "split brain" failure, in which cluster nodes lose their ability to efficiently interact with one another but continue to run resources. Multiple instances of a resource could be started on different nodes if the cluster simply believed that uncommunicative nodes were down. The impact of split brain varies according to the resource form. An IP address set up on two hosts on a network, for example, can cause packets to be sent to one or the other host at random, making the IP address useless. If the user is dealing with a database or a clustered file system, the effect could be much more severe, causing data corruption or divergence.

4.5.1 Fencing Device

A fence device (also known as a fencing device) is a resource that allows users to fence a node. Intelligent power switches and IPMI systems that accept SNMP command to cut power to a node are examples of fencing devices, as are iSCSI controllers that enable SCSI reservations to be used to cut a node's connection to a shared disc. When fencing devices will be used to recover from the loss of networking access to other nodes, it is important that they do not use the same network as the cluster, as this will result in a single point of failure. Although the loss of a node due to a power outage is indistinguishable from the loss of network access to that node, at least one fence unit for that node must not share power. An on-board IPMI controller that shares power with its host, for example, should not be used as the host's sole fencing unit.

4.5.2 Fencing Agent

A stonith-class assets agent is a fence agent (or fencing agent). The fence agent model defines commands that the cluster can use to fence nodes (such as off and reboot). This, like other resource agent classes, adds a layer of abstraction so Pacemaker doesn't have to worry about complex fencing technology because that information is contained within the agent.

4.5.3 Working Principle

Usually, fencing devices only have an interface from which commands can be sent to an external computer. Fencing can also be started at any stage in the cluster life cycle, even before any resources have been started, by Pacemaker, other cluster-aware applications like Distributed Lock Manager(DLM) or manually by an administrator. Pacemaker does not need the fence system resource to be "started" in order to be used, in order to accommodate this. Whether or not a fence device is started decides whether or not a node operates any repeated monitors for the device, as well as giving the node a small advantage of being selected to perform fencing for that device. Any node can run any fencing system by default. When a fence device's target-role is set to Stopped, it can no longer be used by any node. If a node's mandatory location constraints prohibit it from "running" a fence device, it will never be chosen to perform fencing with the device. A node can fence itself, but the cluster can only do so if no other nodes are capable.

5 Case Study - Borg

5.1 Introduction

Borg [16], Google's own cluster management system, runs thousands of jobs per second and supports a wide variety of applications across hundreds of clusters. A declarative task description language, real-time job tracking, and tools to evaluate and simulate system behaviour are some of the core features that Borg provides to its users. Borg's main advantages are that it abstracts away the details of resource management and failure handling, allowing users to concentrate on application development and writing logic for their jobs/services. It is highly fault tolerant, ensures high availability and reliability for the applications that run on it.

5.2 Architecture

Borg is composed of multiple cells where each borg cell consists of a set of machines, a logically centralized controller called the Borgmaster, and each machine in the cell runs a borg agent process called the Borglet as shown in Fig 6.

5.2.1 Borgmaster

Each cell's Borgmaster consists of two processes: the main Borgmaster process and a separate scheduler (more on that in section 5.2.3). It coordinates all the activities in the cell. Since a single master can fail, every cell has five replicas of the Borgmaster as a fault handling mechanism. When one of the

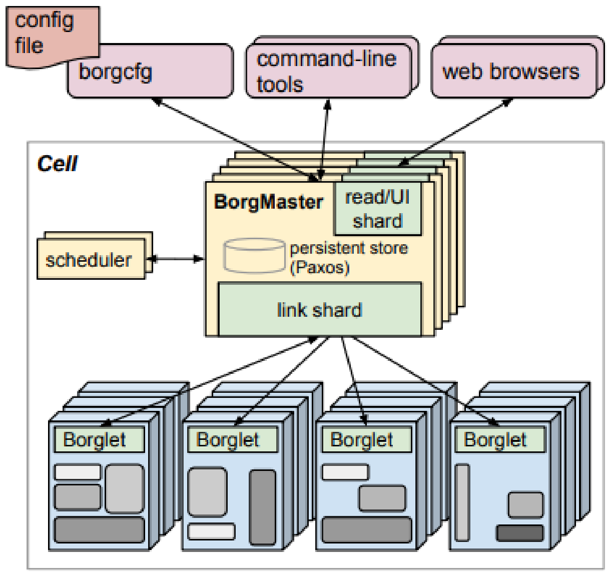

Figure 6: The high-level architecture of Borg: [16]

replicas fails, a leader election is held, and the primary Borgmaster, to whom the consumer submits jobs, is chosen. The Borgmaster takes a snapshot of its current state on a regular basis and saves it in secure storage.

5.2.2 Borglet

Borglet is a cell-wide process that runs on all nodes. It starts and stops tasks, restarts them if they fail, manages local resources, and reports on the machine's status to the Borgmaster. Borgmaster polls each Borglet every few seconds to determine the current state of the system and then sends additional requests to the machines. Borglet will continue to run even if all Borgmaster replicas fail, ensuring that currently running tasks and services remain available.

5.2.3 Scheduling

In this section, we will go over all that happens in Borg from the time a job is submitted to the time it is done. A client submits the work to the Borg system, along with the job's name, his name, and the number of tasks in the job. If the job can be

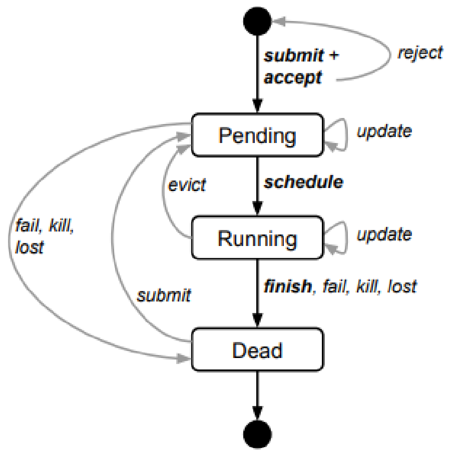

Figure 7: State diagram for both jobs and tasks : [16]

admitted based on its quota, the Borgmaster notes it in its secure storage and adds it to the scheduler queue. The scheduler sorts the jobs in the queue from highest to lowest priority and chooses the topmost job to be dispatched for execution. The Borglet takes over the dispatched task and manages everything relevant to the task. The Borglet also sends the Borgmaster status updates on a regular basis. The Borglet tells the Borgmaster when the task is completed, and the Borgmaster makes the required adjustments to its secure storage, updates the state of the computer that completed the task, and reports the results of the task to the client.

5.3 Design Goals

5.3.1 Job Submission

In Borg, a job is sent to a single Borgmaster. Before doing something else, the Borgmaster makes a note of the job, since it will be responsible for all aspects of it. The job being submitted can be a list of tasks that can be scheduled individually on various machines.Also, after the job has been submitted, the client can still make changes to it.

Since the entire scheduling process is controlled by a single server, load balancing can be optimised and scheduling overheads are kept to a minimum. Owing to the lack of a distributed submission, communication overheads are significantly reduced. It is much simpler to provide the consumer with a global view of the system at any time.

But because the Borgmaster does not share the scheduling load with any other nodes, communication links of Borgmaster are often more crowded which can cause network congestion.

5.3.2 Priority, quota, and admission control

Each job that is submitted to the Borg system has a priority assigned to it. Jobs are scheduled in order of importance, from high to low. Using a Round-Robin scheme based on arrival times, a total order is imposed among jobs with the same priority. If a higher priority task cannot be scheduled in the system when a lower priority task is being completed, the former can prempt. Preempting a production task for another production task is not permitted to stop cascading. In addition to priority, the user often requests a certain amount of resources known as quota. According to its quota, Borg accepts a work for scheduling. It is possible that the user's work requirements after admission would surpass this quota at some stage. In such instances, Borg reduces the job's priority while ensuring that the product of the priority and modified quota is constant at all times.

The priority system helps high-priority jobs to be done more efficiently than low-priority jobs. The priority and quota-based scheme ensures that the user's job is always completed, regardless of whether the user has the right estimate of the amount of resources available for his job.

But in some cases low-priority jobs can go unfilled due to incoming high-priority jobs; they might not be scheduled or even if they are scheduled, they are constantly preempted. Also in certain situations, low priority jobs may be forced to give up a portion of their shareable resources to high priority jobs, affecting their latency.

5.3.3 Scheduling and Preemption

The scheduler takes over after the Borgmaster receives a job. The scheduler only works on individual tasks not jobs. It attaches the tasks to its pending queue and prioritises them before dispatching them. The main components of the scheduling algorithm are feasibility testing and scoring. Feasibility testing entails locating machines that meet the task's constraints and have all of the services needed. It's worth mentioning that when measuring available resources, Borg often considers the resources already allocated to low-priority tasks, since these could be preempted to make room for the higher-priority task. After determining the machines that are feasible, the "goodness" of each machine is calculated. This stage is referred to as scoring. The scores are certainly influenced by the user's preferences, but other factors include reducing the number of preempted tasks, selecting machines that already have the packages needed by the task mounted, spreading tasks across power and failure domains, and evenly distributing high and low priority jobs across all machines. Both best-fit and worst-fit strategies are used to generate a ranking, each with its own set of advantages and disadvantages. The need to score all of the available machines is eliminated with a two-stage scheduling policy. Compromising on the user's expectations increases the overall performance of the system substantially.

Even if a component of the system fails, operating on tasks rather than jobs means that only a portion of the job is impacted. The system is better equipped in the event of a fault because activities are distributed across power and fault domains.

Low-priority jobs, on the other hand, can go starving. And even if they are scheduled quickly, when they are preempted, their turnaround times can be very long.

5.3.4 Fault Tolerance

The heartbeat message sent by the Borgmaster to the Borglets is entirely responsible for fault detection and recovery. All of the Borgmaster replicas communicate in the same way. When a Borglet crashes, the Borgmaster detects it and reschedules all tasks that were running on it based on the most recent state for the crashed computer. If the Borgmaster fails, however, the replicas elect a new leader from among themselves. The system's condition is restored after the new elected leader meets with all of the Borglets . As previously mentioned, the Borgmaster takes a snapshot of its current state on a regular basis and saves it in a safe storage. This checkpoint can be used by the new leader to restore the Borgmaster's state (if it has skipped any recent changes to the old Borgmaster).

Since the odds of all the Borgmaster replicas crashing at the same time are extremely slim, almost all forms of crash failures are treated. Periodic checkpointing reduces the amount of time it takes to recover. One significant benefit is that a fault in one part of the system has no effect on the activities in the other parts, which will continue to be completed. As a result, in this model, spreading tasks (of the same job) through machines makes the job fault tolerant to a large extent.

However, upgrading the Borgmaster replicas wastes a lot of computing power and time. The checkpoints take up a lot of space, and most of the time they're useless because device failures are extremely rare.

6 Conclusion

Clusters are used practically by all research labs for their high performance computing needs. Moreover, most of the supercomputers are infact clusters of several smaller but powerful computers. Cluster middlewares manage the resources. They provide an interface for the user to interact with the cluster, giving an abstracted view to the user so that user does not need to worry about the intricacies such as resource allocation etc. Like any distributed system, clusters are susceptible to faults and so to provide a dependable system to the user, there must be sufficient protocols to tolerate faults.

We have elaborated on the several design issues and corresponding design choices and architectures. We have also extensively described three common cluster management systems, SLURM, Borg and Linux-HA. While SLURM is a more customizable and flexible cluster manager making it very common among small clusters in research facilities. Borg is the cluster management system for the clusters at google. Google has large scale clusters with several simultaneous job requests. Linux-HA is a more commercial cluster management tool.

References

[1] Cisco server architectures. https://www.cisco.com/c/en/us/td/docs/solutions/Enterprise/Data_Center/DC_Infra2_5/DCInfra_1.html.

[2] Corosync. https://www.alteeve.com/w/Corosync.

[3] Hawk. https://documentation.suse.com/sle-ha/11-SP4/html/SLE-ha-all/cha-ha-config-hawk.html.

[4] Linux-ha. http://www.linux-ha.org/wiki/Main_Page.

[5] TOP 500. Top 500 november 2020. https://www.top500.org/lists/top500/2020/11/.

[6] Remzi H. Arpaci-Dusseau and Arpaci-Dusseau Andrea C. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, LLC, 1.00 edition, 2015. http://pages.cs.wisc.edu/~remzi/OSTEP/.

[7] ClusterLabs. Pacemaker. https://clusterlabs.org/pacemaker/doc/en-US/Pacemaker/2.0/html-single/Pacemaker_Explained/.

[8] Fujitsu. Fugaku specification. https://www.fujitsu.com/global/about/innovation/fugaku/specifications/.

[9] Team Fujitsu. Fujitsu specification. https://www.fujitsu.com/global/about/innovation/fugaku/specifications/.

[10] IBM. Summit specification. https://www.ibm.com/thought-leadership/summit-supercomputer/.

[11] IIT KGP. Param shakti. http://www.hpc.iitkgp.ac.in/.

[12] Alan Robertson. Linux-ha heartbeat system design. In *4th Annual Linux Showcase & Conference (ALS 2000)*, Atlanta, GA, October 2000. USENIX Association.

[13] Schultz Rod. Slurm basic configuration and users. https://slurm.schedmd.com/slurm_ug_2011/Basic_Configuration_Usage.pdf?fbclid=IwAR051AlhfpLyxNFEPGM47SVDTZ3Oqq5TDYsl6JLwGg

[14] T. Sterling. *Condor: A Distributed Job Scheduler*, pages 307–350. 2001.

[15] SLURM Team. Slurm workload manager documentation. https://www.slurm.schedmd.com/.

[16] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, New York, NY, USA, 2015. Association for Computing Machinery.

[17] Carl A. Waldspurger. Memory resource management in VMware ESX server. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 181–194, 2002.

[18] Tim Wickberg. Introduction to slurm. https://slurm.schedmd.com/SLUG17/SlurmOverview.pdf, 2017.

[19] Andy B. Yoo, Morris A. Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, pages 44–60, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.