CMPSCI 677 Operating Systems **Spring 2022**

Lecture 7: February 16

Lecturer: Prashant Shenoy *Scribe: Shruti Jasoria (2022), Deokgi Hong (2019), Ao Liu (2018)*

7.1 Overview

In this lecture, we continued Cluster Scheduling and started a new topic, Virtualization.

7.2 Cluster Scheduling

Cluster Scheduling is scheduling tasks on a pool of servers. This differs from distributed scheduling on workstations. We assume:

- Machines are cheap enough that looking for free CPU cycles on an ideal workstation is not required. There is a dedicated pool of servers to run the tasks.
- Servers are more computationally more powerful than workstations.
- The users explicitly submit the jobs on the cluster.

Cluster scheduling can be analyzed with respect to two applications:

- **Interactive Applications**
  These are applications where user has sent a request and is actively looking for a response. In this case response time is an important factor in designing the scheduler, e.g., for web services.
- **Batch Applications**
  These are long running computations which require powerful machine. In this case the throughput is optimized, e.g., for running simulations.

7.2.1 Typical Cluster Scheduler

Typically, there are two kinds of nodes in a cluster - dispatcher and worker. A request (or task) follows the following process:

1. The user submits a request to the dispatcher node.
2. The dispatcher node places the task on a queue.
3. When a worker node is available, the dispatcher removes the task from the queue and assigns it to a worker node.
4. The worker node runs the task.

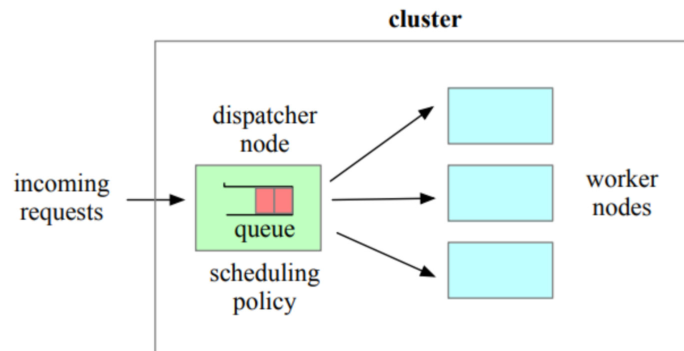

Figure 7.1: A typical cluster scheduler

This design is similar to a thread pool that has listener and worker threads. The scheduling policy depends on the kind of application that has to be run on the cluster.

Question: Would the dispatcher act as single point of failure?
Answer: That would always be a case. To make the system failure tolerant it has to be ensured that in case the dispatcher fails another machine can take over. [1]

7.2.2 Scheduling in Clustered Web Servers

Interactive applications like websites that receive large number of requests are hosted on a cluster. Consider a cluster of N nodes. Here one node acts as the dispatcher, called the *load balancer*. The $(N-1)$ worker nodes are running a replica of the web server and act as the server pool. Any of the worker nodes can service the user requests. The dispatcher receives the request from the user and directs it to one of the worker nodes. The scheduling of the user requests can be done on the basis of:

- The dispatcher can pick up the **least loaded** server.
- The dispatcher can schedule the job in a **round robin** manner. In this case it would not need to get periodic load information updates from the worker nodes.
- If the worker nodes are heterogeneous, **weighted round robin** can be used where you weight a node by the amount of resources and allocate more requests to machines with more computation power.

In case of stateful applications, for eg. an online shopping store the state of the shopping cart is maintained by the machine which served the user request. In this case the request level scheduling won't work as the worker node maintains the session for the user. In these cases **session-level load balancing** is used. When a new web browser starts a session round robin is used to allocate the request to worker node. The server is then mapped to the user and all the subsequent requests by the user are sent to the same machine. One way around this is sending the state back to the client as a cookie which the client has to send it back to the server with every request. This way the state is maintained on the client and not the server. This can allow request level load balancing but would be more expensive as the state updates are sent back to the user.

Question: If you have state, can you keep it in shared memory or database?

[1] A good case study on how this can be achieved is Apache Kafka.

Answer: Shared memory is not a typical OS abstraction. There is no such thing as distributed shared memory that most operating systems support. There can certainly be distributed storage or shared database that can be used to maintain the state. To do so, you'd have to pay the price of I/O to retrieve the state for every request. [2]

7.2.3 Scheduling Batch Applications

This is used for larger jobs like ML training, data processing, simulation. These are long running jobs that can take seconds to as long as hours to finish running. All these jobs come to the dispatcher queue and have to be assigned to a worker node. For this we need a batch scheduler that decides which node to run the job.

One popular batch scheduler available on linux muchines is **SLURM** (Simple Linux Utility for Resource Management). Fundamentally it has a queue and it has a scheduling policy. But it has many other feature:

- It divides the cluster into subgroups. Each subgroup has its own queue.
- This allows user groups with separate scheduling policies for jobs.
- The various queues can put constraints based on runtime of the job. For eg. a queue for short jobs would terminate the job if exceeds the time limit.

7.2.4 Mesos Scheduler

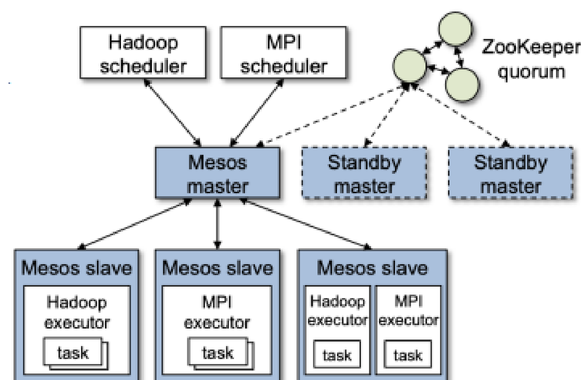

Figure 7.2: Mesos Architecture

The Mesos Scheduler was designed in Berkley but now has been adopted in open source projects. It is a cluster manager and scheduler for multiple frameworks. Mesos dynamically partitions the cluster on the fly based on resource needs of different frameworks. It has a heirarchial two level approach. Mesos allocates resources to a framework and the framework allocates resources to the task. At any point, Mesos keeps taking back resources from frameworks that do not require them anymore and assigns them to frameworks that need them.

Mesos introduces the concept of **resource offer**. Whenever there are idle servers, Mesos creates an offer and sends it to the framework. The framework either accepts or rejects the offer made by Mesos. It accepts the offer if the resources being allocated are sufficient to run the task. Otherwise, it rejects the offer. There

[2] For example, Redis can be used as a distributed in-memory cache because of

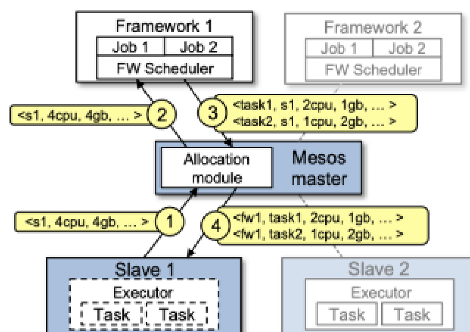

Figure 7.3: Mesos Scheduling

are policies to decide which framework has to be offered the resources. Once the task run by framework is done and the resources are idle, it is available to be re-offered by Mesos.

Question: If there is a busy framework and a not-busy framework, do you want to offer the a not-busy framework more resources?
Answer: The ramework itself is a cluster scheduler. If the framework is busy, it'll take the resources. It is advantageous for the framework to get more resources.

Follow up question: Would a framework accept resources even if it has satisfactory amount of resources?
Answer: A framework accepts resources if it has pending tasks in its queue. The frameworks are designed to be well-behaved, i.e., they will not accept resources if they don't need them.

Question: Once the resources are free, how are the sent back to Mesos master?
Answer: The worker nodes have a mesos monitor running on them. Once the node is idle, that entity returns the node to Mesos master.

Question: Does Mesos have any intervention if the client contacts the server?
Answer: From the client's perspective, Mesos is completely transparent. The client submits the jobs to the queue of relevant framework.

Question: What is the advantage of Mesos?
Answer: The pool of servers allocated to a framework can be dynamically changed at a server granularity. This allows adjustment of resources based on the load on each framework. This makes better utilization of servers in the pool.

Question: Does the Mesos coordinator know resource requirement of each framework?
Answer: No, it doesn't. That is why it makes offers. Mesos makes its decision based on acceptance or rejection of the offer. If the offer is rejected it could be one of the two situations. Either the framework doesn't require more resources or the resources offered are not enough. Mesos would probe and try to figure out the situation. It's a "push" approach.

Question: When a task finishes the idle server goes back to the master. Is it a good idea to give up the resource as another task could be received by the framework and that could have run on the idle node?
Answer: Server allocation is done on a per-task basis. If a task is waiting, the framework would eventually get resources assigned.

7.2.5 Borg Scheduler

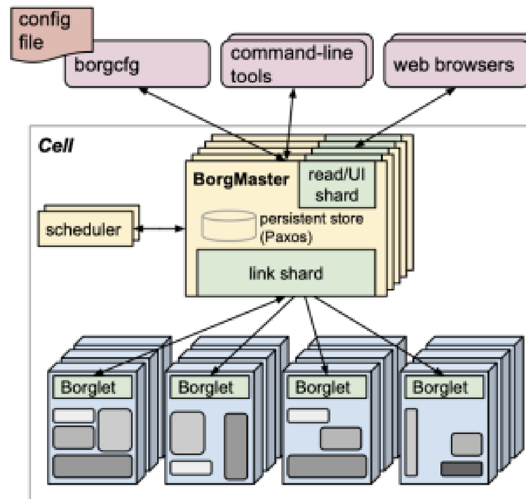

Figure 7.4: Borg Scheduler

This is Google's cluster scheduler that was designed with the following design goals:

- The culture should scale to hundreds of thousands of machines.
- All the complexity of resource-management should be hidden from the user.
- The failures should be handled by the scheduler.
- The scheduler should operate with high reliability.

These ideas were later used to design Kubernetes. Borg is designed to run both interactive and batch jobs. Interactive jobs like web search, mail are user facing production jobs are given higher priority. Batch jobs like data analysis are non-production jobs are given lower priority. Interactive jobs will not take up the entire cluster. The remaining cluster is used for batch jobs.

In Borg, a **cell** is a group of machines. The idea behind Borg scheduler is to match jobs to cells. The jobs are going to specify the resource needs and Borg will find the resources to run it on. If a job's needs change over time, it can ask for more resources. Unlike Mesos, the jobs do not wait for an offer but "pull" the resources whenever they need it.

Borg has built-in fault tolerance techniques. For e.g. if a cell goes down, Borg would move the job to some other cell. To allocate set of machines to a job, Borg would scan machines in a cell and start reserving resources on various machines. This is called **resource reservation**. The resources reserved are used to construct an allocation set. This allocation set is offered to the job.

Question: So instead of waiting, the framework can ask for more resources?
Answer: There is no notion frameworks in case of Borg. A job can ask for more resources. A job itself could be framework, batch job or a web application. So Borg is not necessarily doing a two-level allocation.

There is ability to preempt. Lower priority jobs are terminated to allocate resources to a higher priority job.

Question: When you say terminate, are we talking about pausing a job or killing a job?
Answer: Termination is a policy decision. The important part of preemption is to reclaiming the servers. If the job being terminated is a web server, it doesn't make sense to pause it. But if is a long running batch job, it would make sense to pause it.

Question: Is the priority determined based on interactive and batch or can we have multiple priorities?
Answer: There is a clear distinction between interactive and batch. The idea can be extended to have multiple levels. Kubernetes allows you to make multiple priority levels.

The coordinator of Borg is replicated 5 times for fault tolerance. The copies are synchronized using Paxos(would be studied in greater detail in Distributed Consensus). It is designed with high degree of fault tolerance.

The scheduler has a priority queue. So the scheduler can either be best-fit or worst-fit depending on whether you want to spread the load or concentrate the load.

7.3 Virtualization

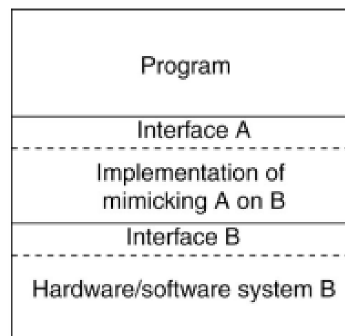

Figure 7.5: Virtualization

Virtualization uses or extends an existing interface to mimic the behaviour of another system. It is introduced in 1970s by IBM to run old software on newer mainframe hardware. It provided a software layer which emulates an old hardware, so an old software could be run on top of the software layer.

Question: Do mainframe machines have an operating system?
Answer: Yes, OS/360 and many generations of it. Early versions of UNIX came out from those OSes.

Question: What part of the image is the software layer?
Answer: The part between Interface A and B. That is the virtualization layer and it is implemented in software.

7.4 Type of Interfaces

Using virtualization, different layers of the hardware-software stack can be emulated.

- Assembly instructions (Hardware virtualization)

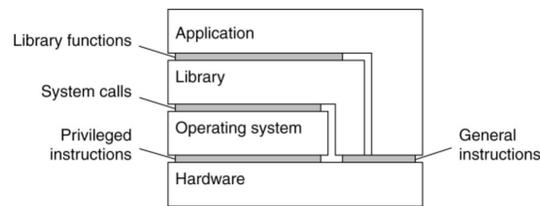

Figure 7.6: Types of interfaces in virtualization

- System calls (OS-level virtualization): E.g., The open-source software Wine emulates Windows on Linux.
- APIs (Application-level virtualization): E.g., JVM

7.4.1 Emulation

In emulation, a virtual machine (VM) fully emulates every aspects (CPU, a disk, a network interface card) of other machine. It is a software simulation of hardware. This is the most flexible technique as it can emulate any hardware on any hardware. But it's also the slowest as every machine instruction has to be translated and emulated.

Examples: Bochs, VirtualPC for Mac, QEMU.

7.4.2 Full/Native Virtualization

The VM does not emulate the entire machine, but it emulates enough hardware to run another system. This is typically done when an emulated interface and a native interface are the same.

Applications of nature virtualization include:

- Running a completely different OS on top of a host OS.
- Acting as a sandbox for testing.
- Running multiple smaller virtual servers on a single server.

Examples: IBM VM family, VMWare Workstation, Parallels, VirtualBox.