Chapter Overview - YARN

Yet Another Resource Negotiator

YARN - Why?

In this session, we are going to discuss YARN - Yet Another Resource Negotiator. YARN is a resource manager which keeps track of various resources such as memory and CPU of machines in the network. It also runs applications on the machines and keeps track of what is running where.

Before jumping into YARN architecture, let try to understand with an example why we need distributed computing

Let us say we have a computer with 1 GHz processor and 1 GB RAM. It takes 20 milliseconds to read the profile pic from disk and then 5 more mill seconds to resize it. How much time would this computer take to resize a million profile pics?

Can we do two things in parallel when dealing with so many pics? Yes because reading from disk involves mainly the disk and resizing mainly involves CPU and RAM. So, reading and resizing can be done in parallel as shown in the diagram. In the diagram, time is increasing from left to right.

You can see that while pic1 is being resized, pic2 is being read from the disk. For three pics, it takes 20 times 3 plus 5 milli seconds for resizing. Not 25 times 3. So, it took 65ms not 75ms.

So, it is only the disk read time that matters we can completely ignore the last 5ms on large scale. For one million pics it would be 1 million times 20 milliseconds which is approximately 5.5 hours

5.5 hours is not good enough? The next questions is how can we make it faster?

If we use a computer which has four cores or processors, can this process finish in less than 5.5 hours?

No, because it is not the CPU which is causing the delay. The main time is being consumed in disk reads. If we make disk reads faster, the process will become faster. Disk reads can be made faster by using Solid State Drives and by using many disk drives.

YARN - Evolution from MapReduce 1.0

Before YARN, it was MapReduce 1.0 that was responsible for distributing the work. This is how MapReduce 1.0 works. It is made up of a Job Tracker and many task trackers. Job Tracker is like a manager of a shopfloor who is responsible for interacting with customers and get the work done via Task Trackers. Job tracker breaks down the work and distributes parts to various task trackers. The task trackers keep Job Tracker updated with the latest status. If a task tracker fails to provide the status back, Job Tracker assumes that the task tracker is dead. Thereafter Job Tracker assigns work to some other task tracker.

To ensure equal load on all task trackers, Job Tracker keeps track of the resources and tasks.

MapReduce 1.0 also performed the sorting or ordering required as part of Map-Reduce framework.

But MapReduce framework was very restrictive - the only way you could get your work done was by using MapReduce framework. Not all problems were suitable for MapReduce kind of model. Some of the problems can be solved better using other frameworks. That's why YARN came into play.

Basically, Map-Reduce 1.0 was split into two big components - YARN and MapReduce 2.0. YARN is only responsible for managing and negotiating resources on cluster and MapReduce 2.0 has only the computation framework also called workfload which run the logic into two parts - map and reduce. MapReduce 2.0 also does the sorting of the data.

This refactoring or splitting made way for many other frameworks for solving different kind of problems such as Tez, HBase, Storm, Giraph, Spark, OpenMPI etcetera

The advantages of YARN are: 1. It supports many workfloads including MapReduce. 2. Now, with YARN, it became easier to scale up. 3. The MapReduce 2.0 was compatible with MapReduce 1.0. The program written for MapReduce 1.0 need not be modified. Just recompilation was enough. 4. This improved the cluster utilization as different kinds of workloads were possible on the same cluster. 5. It improved Agility. 6. Since map-reduce was batch oriented, it was not possible to run tasks that needed to be run forever such as stream processing jobs.

The role of Job Tracker in MapReduce 1 is now split into multiple components in yarn - Resource Manager, Application Master, Timeline Server. The task tracker is now Node Manager. The role of a slot in MapReduce 1 is now played by Container in YARN.

YARN - Architecture

YARN - Architecture

Let us try to understand the architecture of YARN. The objective of YARN is to be able to execute any kind of workfload in distributed fashion.

All of the components that you see in the diagram are software programs. It has one resource manager. Each machine has a node manager. The node managers create containers to execute the programs. Application Masters are the programs per application that are executed inside the containers. Examples of Application Masters are Mapreduce AM and Spark AM.

The flow goes like this: A client submits an application to the resource manager. The resource manager consults with the node managers and creates an Application Master for the client inside a container.

Now, the application master registers itself with resource manager so that the client can monitor the progress and interact with the application master.

Application Master then requests Resource Manager for containers. The resource manager consults and negotiates with Node Managers for the containers. Node Managers create containers and launch app in these containers. The node managers talk back to the resource manager about the resource usage. To check the status, the clients can talk to Application Master. On completion, Application Master deregisters itself from the resource manager and shuts down.

It can be better understood if we think of YARN as a country where Resource Manager is the president. And clients are organizations like Microsoft or Shell who want their work to be done by the way of establishing offices. Each node is a state such as California and the node manager is governor of the state. The container is the place for the office. The application master is the head office.

Though it looks like over simplification, it would make us understand how YARN works. Microsoft wants to get their offices to be setup, so they contact the president, the president talks to governors of the states and then the head office is established. After wards to get more offices established in various states, head office would request president and the president would discuss with governors and offices would be established.

More On Architecture

YARN - Advanced

What does the resource request made by client or application master have? The request may contain how many containers, how much memory and how many CPU are required. The request may also contain a constraint such as nodes nearer to certain data. This is called data locality constraint. It is similar to A mining company requesting for an office near the area which has mines. The request may either have demand for all of the resources up front or as and when required.

Another question that comes to mind is for how long does an application run? The application lifespan can vary dramatically. Lifespan is categorized into three categories: 1. First, One application per job. This is the simplest case. MapReduce is an example. In this, as soon as the job is over the application ends. 2. Second is One application per workflow or user session. Spark operates in this mode when you launch the interactive shell. So, it remains active during the user's session unless user terminates it. 3. The third one is a long-running application such as a server which runs forever. Examples are Apache Slider or Impala.

If you plan to build your own application, my suggestion would be to first try to use existing frameworks such as MapReduce or Spark.

Second, try to utilize existing tools to Build Jobs such as Apache Slider and Apache Twill. With Apache Slider you can run existing distributed Application such as HBase and Twill can execute any Java runnable.

If you still need to build your own YARN applications, please keep in mind that building from scratch is complex. To start with, you can use Yarn project's bundled distributed shell application example.