IJCSN - International Journal of Computer Science and Network, Volume 6, Issue 6, December 2017
ISSN (Online) : 2277-5420
www.IJCSN.org
Impact Factor: 1.5

Detailed Design of Distributed Resource Manager

[1] M Sai Pradeep; [2] Harish Mamilla; [3] S C Gupta

[1] Department of Computer Science & Engg, IIT-Delhi,
Delhi, 110016, India

[2] Department of Computer Science & Engg, IIT-Delhi,
Delhi, 110016, India

[3] Department of Computer Science & Engg, IIT-Delhi,
Delhi, 110016, India

Abstract - Large-scale data processing is growing rapidly as enterprises are moving towards big data projects. Big enterprises are also maintaining distributed data centers across the globe for disaster recovery and business continuance. After experiencing the success of big data projects, need of running future big data projects on distributed data centers arises. In that case, existing resource management solutions such as Apache YARN or Mesos fails as they still have a centralized resource manager. So for extreme scale data centers or distributed data centers, we need a new generation distributed resource management solution.

Keywords - *Resource Manager, Mesos, YARN, Distributed, extreme scale.*

1. Introduction

Apache YARN and Apache Mesos have gained popularity as a resource management layer for distributed and fault-tolerant computing. Apache YARN overcomes the limitation of first generation Hadoop. Apache Mesos addresses the issue via Resource Offer mechanism. But both the frameworks have a centralized resource manager for allocating resources to applications. Mesos employs a scheduler or Application Master for one category of application while YARN uses Application Master for a particular application. Both of them have improved the scalability issues in Hadoop clusters but centralized RM is still a barrier for extreme scales, the scales that are 2 or 3 orders of magnitude larger than current distributed systems. If the organization has multiple co-operating data centers across the geography, then also the existing model won't work. In this paper, we present the next generation resource manager, which changes the existing centralized resource management. We will discuss how the distributed resource manager works and its detail design architecture.

2. History and Rationale

Commodity server clusters are being used vastly now-a-days as it is very cost efficient. Hadoop, Spark, Storm frameworks run in large commodity hardware clusters. But these frameworks were originally designed to do cluster management, scheduling and running of the tasks in a single monolithic architecture. So as per this architecture, more than one framework can't be run in a given cluster at a time. But any organization's requirement says that multiple frameworks should run alongside each other which is beneficial from economical point of view. To facilitate these requirement a resource management layer is required. A resource is any shared system entity needed for execution by a service and multiplexed by the system between the various services it hosts. CPU time, memory, disk and network bandwidth are all examples of resources [1].

Before going into details of distributed architecture, we will justify the limitations of the existing resource manager such as YARN and Mesos.

2.1 Apache YARN

Apache Yarn is a cluster resource management tool for better resource utilization. Before Yarn, for sharing a large cluster, there was only one way which says that cluster need to be partitioned first and after that different partitions will hold different frameworks. But this doesn't guarantee efficient usage of the cluster resource. In

Hadoop v2.0, Yarn, a resource management layer, was introduced to handle this difficulty of running different computational frameworks in the same cluster. It is a software rewrite which decouples MapReduce's resource management and scheduling techniques from the data processing component.

In a Yarn cluster, two daemons or hosts are the main elements. The ResourceManager is known as the master daemon and NodeManager is known as the worker daemon. Earlier in MRv1, there was a single master process, JobTracker and a number of subordinate processes called TaskTrackers. JobTracker co- ordinates all the jobs running on cluster and assigns tasks on the TaskTrackers and TaskTrackers run tasks and send reports periodically to the JobTracker. In this architecture, major limitation involves scalability bottleneck which is caused by a single JobTracker. In Yarn, goal was to eliminate this limitation by creating a short-lived JobTracker known as ApplicationMaster.

In Yarn architecture, ResourceManager arbitrates the available cluster resources. It tracks the number of live nodes and resources available on the cluster. When a client submits an application, an instance of ApplicationMaster is started to coordinate all task's execution within that application. Therefore, ApplicationMaster becomes responsible for monitoring tasks, restarting failed tasks etc. which was earlier handled by the single JobTracker. NodeManager has a number of dynamically created resource containers of different sizes e.g. RAM, CPU. It also manages processes running in containers. Containers do run different types of tasks which includes ApplicationMaster also. The YARN configuration file is an XML file that contains properties. This file is placed in a well-known location on each host in the cluster and is used to configure the ResourceManager and NodeManager. [2]

It can be easily concluded from the above architectural discussion that ResourceManager is centralized and can become bottleneck for the next generation extreme-scale data centers.

2.2 Apache Mesos

Mesos is built using the same principles as the Linux kernel, only at a different level of abstraction. The Mesos kernel runs on every machine and provides applications (e.g., Hadoop, Spark, Kafka, Elastic Search) with APIs for resource management and scheduling across entire datacenter and cloud environments.

Basically, Mesos has a master process and set of slave processes which runs on the cluster nodes. On slave nodes, different computational frameworks are being deployed on top of Mesos and they run individual tasks on them. Master process has the responsibility of resource sharing using resource offers to the frameworks. Mesos master process uses resource allocation policies and available free resources for allocating resources to computational framework.

In this architecture, Mesos does not need frameworks to specify their resource requirements. Frameworks can reject the given offers. A framework reject resource offers which do not meet requirements and also can wait for the satisfied ones. This leads to a limitation that Mesos may send too many offers before the accepted ones. To minimize this limitation, Frameworks can set filters using which certain resources won't be offered by Mesos master.

Mesos running frameworks have to implement a resource scheduler and an executor process. Mesos master offers resources to the resource scheduler so the framework resource scheduler has to register with master node. Executor processes runs on cluster nodes and can run individual tasks on these nodes. [3]

From the above discussion, it can be easily seen that Mesos has a single point of failure in master node. But Mesos uses Zookeeper service to elect a new master in case of master failure. Still only one master be active at any time which can be a bottleneck.

2.3 Limitations

Now-a-days every corporation is opting for distributed data centres. There are several reasons for opting this solution. One of the primary reason is disaster recovery and business continuance. Data depositary is also getting distributed across data centres which leads to high availability of applications and data access. It also helps in load balancing and performance scalability. Current resource management layer solution i.e. Apache YARN and Mesos doesn't address this kind of data centres as it has a centralized resource manager. YARN and Mesos decoupled resource management with the programming model which leads to unprecedented scalability compared to Hadoop version 1. But Centralized RM prevent Hadoop from scaling to extreme scales which are 2 or 3 orders of

magnitude larger than current distributed systems. Our proposed design will address extreme scale issue as well as distributed data centre issue.[4]

3. Logical Architecture

Apache YARN and Mesos are known to scale up to about few thousands of machines. Let's refer this as a sub-cluster. Extreme scale can be referred as tens of thousands of nodes which will become a cluster comprising of all sub-clusters. A master will be assigned for each sub-cluster so that multiple masters will take care of the whole cluster. All the masters will co-operate with each other using Gossipers. A single policy maker needs to be introduced for controlling corporate policies.

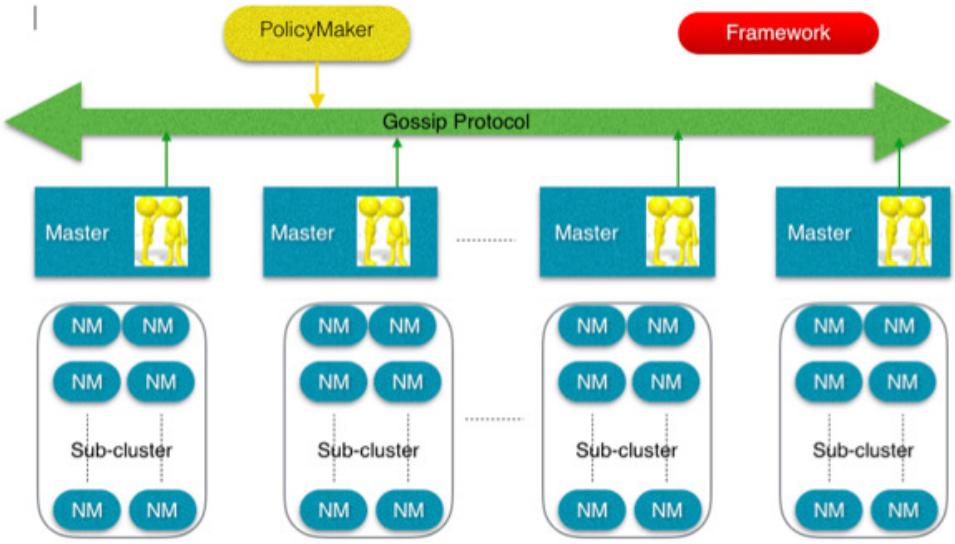

Fig. 1 Logical Architecture

In this section we will discuss high level architecture and components of the proposed design.

3.1 Sub-Cluster

A sub-cluster consists of a number of commodity workstations or PCs connected by high speed network. It contains up to few thousands nodes. The exact size of the sub-cluster can be determined by considering best practices and ease of deployment.Sub-clusters can be geographically distributed and communicate over internet. Sub-cluster is the scalability unit i.e. we can scale out the cluster by adding one or more sub-clusters. By design, each sub-cluster is a fully functional resource management unit.

3.2 Gossiper

Each master is accompanied by a gossiper who is responsible for the whole sub-cluster in the whole cluster. Gossipers talk to each other using gossip protocol. They exchange resource information among themselves. They understand the corporate policy from Policy maker. Resource allocation to the application can only happen after gossipers talk to each other and reach a conclusion and instructs the specific master to do the needful. If some site or sub-cluster is out of resources, then corresponding gossiper will send this information to all other gossipers.

3.3 Policy Maker

Policy maker overlooks the entire cluster and ensures system is configured and tuned. It is a very light module in which several policies can be set and framework requests will be redirected according to that. It will provide a user interface in which user capacity allocation to sub-cluster mappings and other corporate constraints can be set. Main design point is that cluster availability does not depends on always-on Policy Maker. Policy Maker operates continuously but in out of sync from the

cluster operations and provide us a way to enforce certain requirements such as load balancing, trigger draining of sub-clusters that will undergo maintenance etc. If the policy maker is not available, cluster operations will continue as per last published policies.

3.4 Gossip Protocol

Failure detection and monitoring is essential in distributed fault-tolerance computing. Traditionally it was done via centralised way using a database and all nodes query for information. But it is not practical when large number of nodes are involved. Gossip protocol can be used for solving the problem distributively. Gossip protocol is simple in design. Each participant node sends out heartbeat or some data to other participant nodes. Data propagates thought out the cluster like a virus. After some time duration, data or heartbeat propagates to all the participants. Each participant node maintains a list of known member and an integer, heartbeat counter which can be used for failure detection. Every Tgossip seconds each participant increments its own heartbeat counter and sends to some random known member. Upon receiving the message, the member merges the list with its own list and adopts the maximum heartbeat counter for each participant. If the heartbeat counter has not changes after Tfail seconds, the member is considered as failed. But member is not be forgotten. Failure detector removes a participant member after some Tcleanup seconds which is in general 2 x Tfail. [5]

4. Design

In this section we will be going through the detailed design of the proposed next generation distributed resource manager. Subsection starts with how cluster initializes, then the role of policy maker host followed by the master,gossiper and executor daemon details. Then one subsection details how distributed mutual exclusion is handled while sending sub-cluster offers. Finally one subsection shows the job execution flow of a framework in the cluster.

4.1 Initiation of cluster

All the commodity PCs are connected with a high speed network connectivity will be referred to as slaves as individual. Agent daemon runs on all the slave machines and is managed by the machine which runs master and gossiper daemon. All the machines will be referred to as a sub-cluster. A cluster consists of several sub-clusters and a policy maker host which enforces certain policies to the whole system. Gossiper daemon can be implemented on the same machine as well as other machine. Same machine implementation will result in faster as it involves inter-process communication compared to network communication. Gossiper handles remote requests in the same way as that of master. So Agents should register themselves with master as well as corresponding gossiper.When Agent daemon starts on machines, they share the existing resource information with their masters. Master aggregates the information and sends the total available resource which can be offered to any scheduler to the gossiper. Gossipers share that information among themselves using gossip protocol. And this information keeps on updated periodically among all the gossipers.

Scheduler/Framework can register with any master of the whole cluster. If the scheduler wants to access the other sub-cluster's resources, it has to register with the gossiper as well. Corresponding gossiper will act as a leader for offering remote sub-cluster to the scheduler. Scheduler accepts or rejects the sub-cluster offer. Upon rejection, new offer will be sent subsequently. For remote sub-clusters, gossiper will act as a proxy for scheduler.

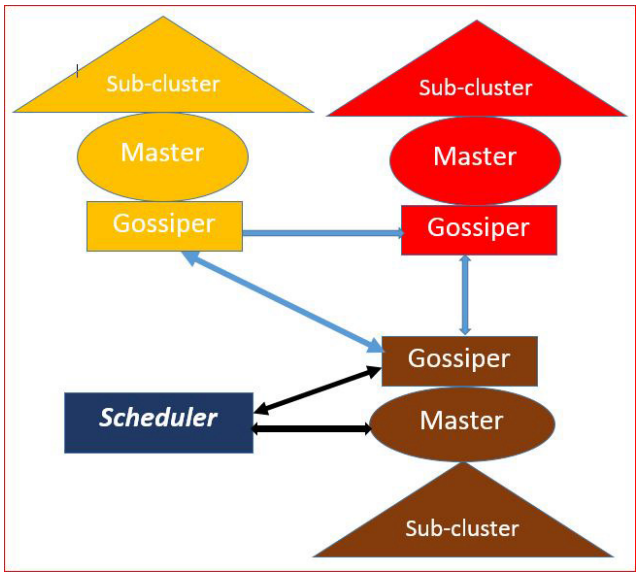

Fig. 2 Scheduler connectivity with Cluster.

All the communications to remote master will be done via registered gossiper. Sub-cluster resources will be offered to the scheduler for executing tasks. Overall Schedulers/Frameworks are going to receive two kinds of

offers i.e. local sub-cluster resource offer and resource offer of the accepted remote sub-cluster.

Scheduler replies to registered master/gossiper with the task information. Master/gossiper sends the tasks to agents.

4.2 Policy Maker

Policy Maker overlooks the entire cluster and ensures system is configured and tuned. It is a very light module in which several organizational policies can be set. When a sub-cluster starts, gossiper should be able to connect to Policy Maker. Once gossiper connects to policy maker, it downloads the required policies to enforce and advertises as a part of cluster to other gossipers.

When multiple masters constitute a whole cluster, failure detection and monitoring is essential. And this requirement should be fulfilled distributively which can be done by Gossip protocol. When a sub-cluster starts, corresponding master will start the gossiper daemon and gossiper registers with policy maker. Policy maker ensures gossip protocol enforcement among all registered gossipers. Every gossiper maintains a list of known gossipers and an heartbeat counter for failure detection.

Policy Maker maintains the membership information of all the registered sub-clusters. Sub-clusters can join or leave independently by notifying policy maker.It offers following APIs.

registerSubCluster() :- Gossiper of sub-cluster uses this API for registration. This is called when the sub-cluster master initialises or restarts.
deRegisterSubCluster() :- This API is used for de-registration of the sub-cluster for maintenance or scaled down purpose.
getSubClusterDetails() :- This API is exposed for providing existing sub-cluster information. This is mainly needed by web UI module.
getAllPolicies() :- This API provides all the existing policies in policy maker. Sub-cluster gossipers use this API for downloading policies.
updatePolicies() :- Sub-cluster gossipers update their policies periodically using this API.
heartbeatDetails() :- This API provides all the heartbeat information of all the sub-clusters.

4.3 Master and Gossiper

Now-a-days In a distributed data center, each sub-cluster master has permission rights enabled for security purposes. Though the higher management allows remote sharing, individual owners of sub-cluster puts restriction on permissions. So from corporate point of view, scheduler may not have permission in accessing all the sub-cluster master hosts.

Scheduler has the choice of registering with any of the master in which it has permission.

SubscribeMaster() :- First request scheduler sends is called SUBSCRIBE-M message which results in a streaming response 200 OK.

SubscribeGossiper() :- Scheduler subscribes to corresponding gossiper with SUBSCRIBE-G message for availing remote sub-cluster offers.

Schedulers need to keep both the connection open as long as possible. All subsequent non-subscribe requests must be sent on different connection and 202 accepted codes will be returned. Subscribe.framework-info.id in the SUBSCRIBE-M/G message helps master/gossiper in deciding new or already subscribed scheduler. Master assigns a new FrameworkID if that field is missing in the message. Gossiper uses the same ID. SUBSCRIBE-M/G message response includes Mesos-Stream-Id header which identifies subscribed scheduler instance.

If the persistent connection opened via SUBSCRIBE-M/G call breaks, master/gossiper considers scheduler as disconnected. Master/Gossiper starts a failover timeout after the disconnection. Scheduler has to re-subscribe within a failover timeout or else master/gossiper considers scheduler as dead and shuts down all executors and tasks. Only one persistent connection will be kept open for a particular Framework ID using Mesos-Stream-Id.

Gossipers send a SC-OFFERS event periodically whenever any aggregated resources become free. This aggregated resource offers of each sub-cluster is shared among all the gossipers periodically via gossip protocol. Distributed mutual exclusion comes into play before sending SC-OFFERS event i.e. only one gossiper can send sub-cluster offer at a time. Offer considered as accepted till accept or decline or offer-timeout period. We will discuss the handling of distributed mutual exclusion in detail in next subsection.

SC-ACCEPT or SC-DECLINE message will be received by the gossiper within the â€"sc-offer-timeout period or else offer stands cancelled. The acceptance offer includes remote cluster details and gossipers are informed by ActionOnAccept() call. Gossiper replies the gossipers with the required API.

ActionOnAccept() :- Gossiper shares the accept offer details with the gossipers and the required sub-cluster gossiper takes action.
ActionOnReject() :- Gossiper informs that it has exited the critical section and sub-cluster offer procedure can be used by another gossiper.

SC-RESCIND event can be sent by the gossiper when a given offer to a scheduler is no longer valid. Any further calls SC-ACCEPT or SC-DECLINE by the scheduler will be discarded. Selected sub-cluster gossiper will have to send the resource offer to the required scheduler via registered gossiper.
ResourceOffer() :- Gossiper sends resource offer after co-ordinating with master but another gossiper acts as a proxy in between scheduler and sub-cluster.

When a resource offer gets accepted, scheduler sends tasks information to the registered gossiper.
LaunchTask() :- Gossiper invokes this API for redirecting this task info to the required gossiper for action.
RegisterAgent() :- Agents register themselves with Gossipers as well as master at the initiation of the cluster.

TEARDOWN message is sent by the scheduler to master/gossiper when it wants to tear itself down. Upon receiving request, master shuts all the executors as well as corresponding tasks in its sub-cluster and gossiper handles the same for remote executors.
Teardown() : - This API is called by gossiper for shutting all the remote executors and its corresponding tasks.

Scheduler has the ability of setting filters on the SC-ACCEPT or SC-DECLINE message for avoiding receiving several unnecessary offers. For removing this filters, SC-REVIVE message can be used by scheduler. KILL message is sent by scheduler to kill a specific task. KILL is forwarded to the required gossiper's executor and executor takes appropriate action and sends TASK-KILLED or TASK-FAILED update. If the task is unknown to the gossiper, TASK-LOST message is generated. Gossiper will release task's resources once it receives the task status update. SHUTDOWN is sent by scheduler to terminate any of the executor. Executor kills all the associated tasks and sends TASK-KILLED updates. executor-shutdown-grace-period is the configured time period in which executor should do the necessary job or else agent will forcefully destroy the container. RECONCILE is sent by scheduler for enquiring of the tasks. Gossiper sends back UPDATE events for each task in the list. HEARTBEAT event is periodically sent by gossiper for ensuring connection is alive.

4.4 Executor

Executor daemons are launched by the agents to run the framework's tasks. The executor daemon interacts with master and gossiper via Agent by subscribing to agent. Following diagram shows the hierarchy.

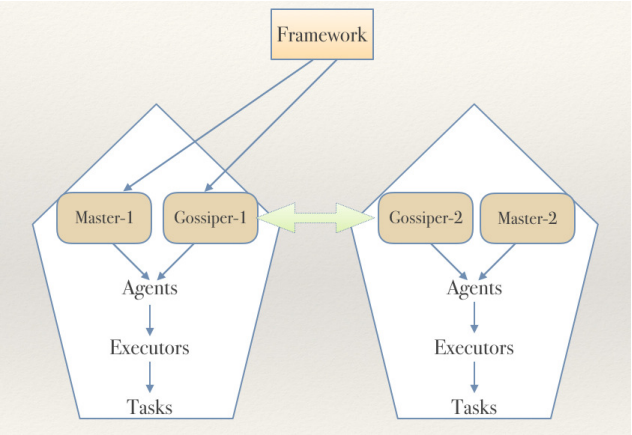

Fig. 3 Hierarchical diagram of major daemons.

SubscribeAgent() :- First request executor sends is called SUBSCRIBE-A message which results in a streaming response of 200 OK. Executors need to keep the subscription request open as long as possible.

All subsequent non-SUBSCRIBE requests must be sent on different connection and 202 Accepted codes will be returned. 202 Accepted response means request is accepted for further processing. If agent reconnects after a disconnection, it sends a list of Unacknowledged status updates using ACKNOWLEDGE events. The executor maintains a list of tasks which are not acknowledged by agent. Executor should subscribe to agent within executor-registration-timeout duration or else agent forcefully destroys executor container. When a task terminates, terminal update should be sent by executor to

agent such as TASK-FINISHED, TASK-KILLED or TASK-FAILED.

Agent sends LAUNCH event to the executor while assigning a new task. Executor sends an UPDATE message which indicates success or failure of the task initialization. If scheduler needs to stop a task, it sends a KILL event. Executor sends an update back to the agent for freeing the allocated resources. Agent can send message to executor upon which executor kills all tasks and sends updates before graceful exit. –executor-shutdown-grace-period is the duration agent waits before forceful termination.

4.5 Handling of DME

Sending sub-cluster offers periodically to the registered frameworks is the responsibility of gossipers. This event is known as SC-OFFERS. The aggregated available resource offers of each sub-cluster is shared among all the gossipers periodically via gossip protocol. The offered resource implies that framework may get maximum of the offered resources which will be further drilled down while offering actual resources.

Distributed mutual exclusion comes into play before sending SC-OFFERS event i.e. only one gossiper can send sub-cluster offer at a time. But this is not the case for master as it manages its own sub-cluster. Offer considered as accepted till accept or decline or offer-timeout period. We will go through in a step-by-step manner to understand how distributed mutual exclusion happens among gossipers before sending any offer.

Cluster consists of n gossipers and each gossiper requires mutual exclusion while giving offer. (n is known via gossip protocol)
When gossiper Gi want to offer, it generates a new timestamp , TS, and sends a message request (Gi, TS) to all gossipers. When a gossiper Gj receives a request message, it may reply immediately or may defer sending a reply back.

When Gi receives a reply message from all gossipers, it sends the offer list to all so that it will be considered as accepted till the end of the offer procedure. After the offer timeout or acceptance, the gossiper sends reply message to all its deferred requests.

The decision whether gossiper Gj replies or defers to a request is based on :

a) If Gj has offered and waiting for response, it defers its reply.

b) If Gj does not want to offer, it sends a reply immediately to Gi.

c) If Gj also wants to offer, it will check TS and if own TS is greater that Gi, then it sends reply or else defer.

The above mentioned concept can be depicted in the following diagram.

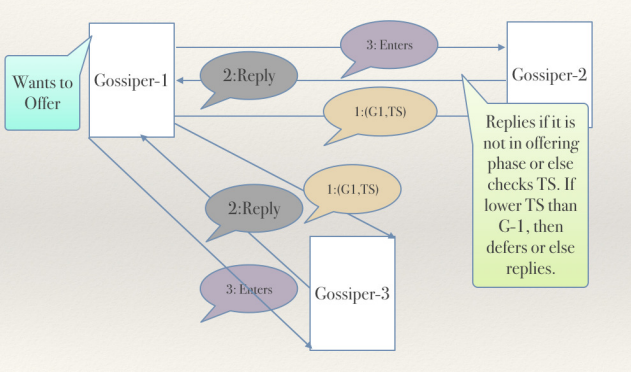

Fig. 4 DME handling among Gossipers.

4.6 Allocation Module

While slaves continually advertises available resources to its master, allocation module is responsible for determining which frameworks should receive a given offer. Allocation module can be made pluggable so that a customer can implement its own allocation mechanism according to business requirement. Default allocation module can include Dominant Resource Fairness (DRF) algorithm. DRF algorithm is a generalization of max-min fairness to multiple resource types. Researchers showed that DRF is fair for multi-tenant systems, Strategy-proof i.e. tenant can't benefit by lying and Envy-free i.e. tenant can't envy another tenant's allocations. Also DRF is usable in scheduling VMs in a cluster.[6]

Further we can fine-tune resource scheduling without replacing or re-implementing the default allocation module. These can be done using roles, weighs and reservations. By combining roles, weights and reservations, guarantee can be provided for specific

applications about availing the cluster resources in a controlled manner.

The concept of roles allows to organize frameworks and resources into arbitrary groups. To use the concept of roles in a given cluster, configuration needs to be done with master and gossiper with a static list of all acceptable roles that will exist across the cluster. By setting a value for the –roles configuration option, e.g. roles: prod, test, remote , frameworks are allowed to register with three common roles - production, testing, remotely available resources.

This allows multiple teams to share a large cluster instead of creating several smaller clusters. Roles can be used for ensuring a specific type of workload runs on only a subset of machines.

In addition to roles, cluster can be configured with weights per role as a means to provide priority to certain roles over another. Using the above example, master and gossiper can be configured to prioritize remote role above that of production and testing.

e.g. weighs : prod=20, test=10, remote = 40

In practice, above rule specifies frameworks in remote role will be offered two times as many resources in the production role. When a new resource offer is advertised to the master, the allocation module checks the roles on the cluster to determine which one is furthest below its weighted fair share. Then the allocation module will check the frameworks within the role and offer resources to the framework that is furthest below its fair share.

Reservations guarantee that certain roles always receive a certain amount of slave's resources. But it may lead to overall decreased cluster utilization.

Suppose we have a single machine with 32 CPUs, 64 GB RAM and 1 TB disk. And we like to ensure half of the resources on the machine i.e. 16 CPUs, 32 GB RAM and 512 GB disk are always available to frameworks registered with the production role. This can be achieved with following configuration on slave:

–resources="cpus(prod):16; mem(prod):32768; disk(prod):524288"
–resources="cpus(*):16; mem(*):32768; disk(*):524288"

The remaining resources are assigned to default role (*) and offered to frameworks that didn't specify a specific role. Following diagram depicts the above concept.

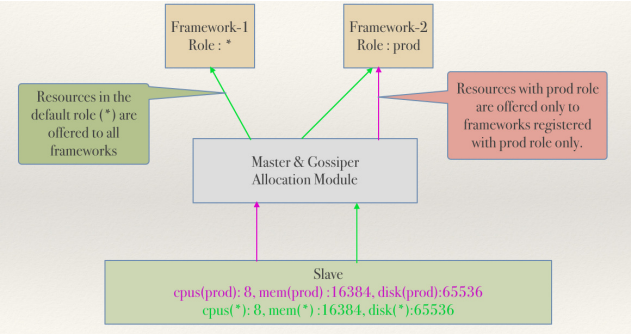

Fig. 5 Allocation module concept

4.7 Job execution flow

The figure below shows an example of how a framework gets scheduled to run a task.

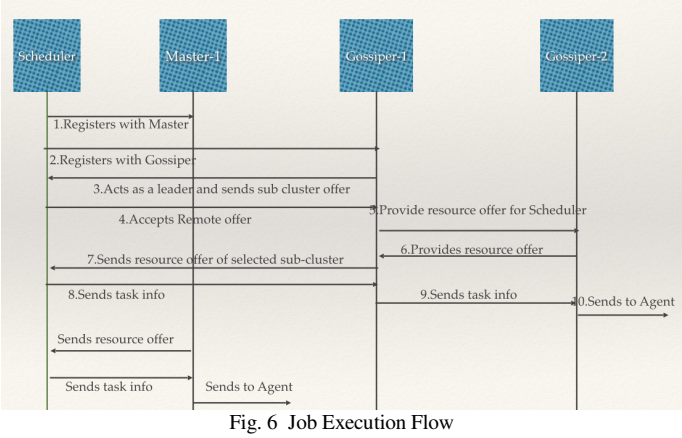

Fig. 6 Job Execution Flow

Suppose there are two sub-clusters each having two agents/slaves. Agents report to their registered master and registered gossiper that each have 4 CPUs and 4GB memory free. Then sub-cluster-1 and sub-cluster-2 has total 8 CPUs and 8 GB of memory free. Gossipers share this information with each other. Let's walkthrough the events.

a) Framework-1 registers with master-1 and Framework-2 registers with master-2.

b) Framework-1 wants to access local sub-cluster resources as well as remote sub-cluster resources. So it registers with gossiper-1. Framework-2 wants to have only local sub-cluster resources. So it does not register with gossiper-2.

c) Gossiper's learn allocation policies from the policy maker. Let's assume gossiper-2 has learned that only 75% of its available resources can be shared remotely. Via gossip protocol, gossiper-1 learns that sub-cluster-2 can offer maximum of 6 GB memory and 6 CPUs. Gossiper-1 acts as a leader and sends offer to framework-1.

d) Framework-1 accepts the remote sub-cluster-2 offer. It sends SC-ACCEPT message to the registered gossiper-1.

e) Gossiper-1 requests gossiper-2 for providing resource offer to framework-1 as it will act as a proxy for all the communications.

f) Master-2 checks via allocation module and decides which resources can be offered to remote framework. Same information is shared to gossiper-1 via gossiper-2. Let it be 4 GB memory and 4 CPUs.

g) Framework-1 receives the resource offer from sub-cluster-2 via the registered gossiper-1.

h) Framework-1's scheduler replies to the gossiper with information about two tasks to run on remote sub-cluster: using (1 CPUs, 1 GB RAM) for the first task, and (1 CPUs, 2 GB RAM) for the second task.

i) Gossiper-1 sends the tasks information to the required gossiper in the sub-cluster i.e. gossiper-2 in this case.

j) Gossiper-2 sends the tasks to the agent, which allocates appropriate resources to the framework's executor, which in turn launches the two tasks. Now, (2 CPUs, 1 GB RAM) is added to the available resources.

k) Master-1 offers local sub-cluster resources to Framework-1 using allocation module. Framework-1 has the choice of accepting or rejecting offer.

l) If it accepts the offer, then it sends tasks info along with ACCEPT message. And master sends the tasks to agent.

4. Conclusions

We have discussed a distributed resource management layer solution which allows distributed as well as extreme scale data centers to share resources in an efficient and controlled manner. Existing resource manager solutions such as YARN and Mesos does not address the distributed and extreme scale data centers issues as they have a centralized host to manage resources. Our solution distributes that module so that centralized RM will not be a bottleneck. It can be easily scalable by adding a new sub-cluster. Policy maker host manages the whole cluster but sub-clusters are not dependent on the always-on policy maker host. Data center requirements such as load balancing, trigger draining of sub-clusters that will undergo maintenance etc. can easily be handled by enforcing policies via policy maker. If the policy maker is not available, cluster operations will continue as per last published policies. Together these elements make our solution feasible to all distributed and extreme scale data centers.

References

[1] Mohit Aron, Peter Druschel, Willy Zwaenepoel, Cluster Reserves: A Mechanism for Resource Management in Cluster based Network Servers, Rice University

[2] Vinod, Arun, Chris, Sharad, Robert, Thomas, Jason, Carlo , Apache Hadoop YARN : Yet Another Resource Negotiator, SoCC13, 13 Oct. 2013, Santa Clara, California, USA. ACM 978-1-4503-2428-1.

[3] Benjamine et al. Hindman, Mesos: A platform for fine grained resource sharing in the data Centre, in Proceedings of the 8th USENIX conference on Networked systems design and implementation,2011

[4] Ke Wang , Ning Liu , Iman Sadooghi , Xi Yang , Xiaobing Zhou , Tonglin Li , Michael Lang , Xian-He Sun , Ioan Raicu, Overcoming Hadoop Scaling Limitations through Distributed Task Execution, in 2015 IEEE International Conference on Cluster Computing

[5] Robbert van Renesse, Yaron Minsky, and Mark Hayden, A Gossip-Style Failure Detection Service , , Dept. of Computer Science, Cornell University 4118 Upson Hall, Ithaca, NY 14853

[6] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, Dominant Resource Fairness: Fair Allocation of Multiple Resource Types , Ion Stoica University of California, Berkeley

[7] Arka A. Bhattacharya1 , David Culler1 , Eric Friedman2 , Ali Ghodsi1 , Scott Shenker1, Hierarchical Scheduling for Diverse Datacenter Workloads University of California, Berkeley

[8] Apache Myriad Online [https://www.youtube.com/watch?v=aXJxyEnkHd4]

[9] http://mesos.apache.org/

[10] https://hadoop.apache.org/docs/r2.7.1/hadoop-yarn/hadoop-yarnsite/YARN.html

[11] Cloudera Blog for Yarn

[12] http://www.adaptivecomputing.com/products/open-source/torque/

[13] FENG LI and BENG CHIN OOI, M. TAMER OZSU, SAI WU. 2014. Distributed Data Management Using MapReduce, In ACM Computing Surveys, 2013

[14] Apache Software Foundation (2013 Oct) [Online] : http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop- yarnsite/YARN.html

[15] Apache Software Foundation [Online] : http://www.apache.org/

[16] Jeffrey, and Sanjay Ghemawat Dean, MapReduce: simplified data processing on large clusters, in Communications of the ACM 51.1(2008): 107-113

M Sai Pradeep currently pursuing Masters at Indian Institute of Technology, Delhi. He has completed B.E. from M S Ramaiah Institute of Technology, Bangalore in Computer Science on 2012. He has worked in Samsung R&D, Noida for 1 year and currently working at Indian Oil Corporation Ltd. His research interests includes Big Data, Cloud computing and Data Analytics.

Harish Mamilla currently pursuing Masters at Indian Institute of Technology, Delhi. He has worked in Honeywell Pvt Ltd.