**Global Scheduler Architecture Design Specification**

Table of Contents

[1 Introduction 3](#_Toc49269902)

[1.1 Problem Statement 3](#_Toc49269903)

[1.2 Goals and Scope 3](#_Toc49269904)

[1.3 Design Considerations 3](#_Toc49269905)

[1.3.1 Monolithic Scheduler (Kubernetes, Openstack) 4](#_Toc49269906)

[1.3.2 Siloed Multi-Scheduler 4](#_Toc49269907)

[1.3.3 Shared-State Multi-Scheduler 5](#_Toc49269908)

[1.3.4 Siloed Multi-Scheduler with Intercommunication 6](#_Toc49269909)

[1.3.5 Siloed Multi-Scheduler with Arbitrator 6](#_Toc49269910)

[1.3.6 Two-Level Scheduler 7](#_Toc49269911)

[1.3.7 Summary of Scheduler Design Pain Points 8](#_Toc49269912)

[2 Global Scheduler Overall Architecture Design 8](#_Toc49269913)

[2.1 Key Requirement Analysis 8](#_Toc49269914)

[2.2 Key Points of this New Scheduler Architecture Design 9](#_Toc49269915)

[2.3 High Level Architecture Design 9](#_Toc49269916)

[2.4 Architecture Component Description 10](#_Toc49269917)

[2.5 Key Global Scheduler Mechanism Design 14](#_Toc49269918)

[2.5.1 Shared State Lock-Free Optimistic Concurrent Scheduling Operation 14](#_Toc49269919)

[2.5.2 Multi-Dimensional Optimization Based Scheduling Algorithm 15](#_Toc49269920)

[2.5.3 Dynamic Geolocation and Resource Profile Based Partition Scheme 19](#_Toc49269921)

[2.5.4 Conflict-Aware VM/Container Request Distribution and Scheduling Mechanism 20](#_Toc49269922)

[2.5.5 Smart Conflict Avoidance Approach 21](#_Toc49269923)

[2.5.6 Scheduling Failure 21](#_Toc49269924)

[2.5.7 Scheduling of Scale Set 22](#_Toc49269925)

[2.5.8 Priority Scheduling and Fair Scheduling to Avoid Security Attack 22](#_Toc49269926)

[3 System Flow and Interface Design 22](#_Toc49269927)

[4 Global Scheduler Subsystem Design 23](#_Toc49269928)

[4.1 API Server 23](#_Toc49269929)

[4.1.1 VM/Container Resource Model 23](#_Toc49269930)

[4.1.2 VM/Container Resource API 23](#_Toc49269931)

[4.1.3 Scale set VM/Container Resource API 23](#_Toc49269932)

[4.2 Data Store 23](#_Toc49269933)

[4.3 VM/Container Request Distributor 23](#_Toc49269934)

[4.4 Dynamic Partition Mapper 23](#_Toc49269935)

[4.5 Scheduling Algorithm Executor 23](#_Toc49269936)

[4.6 Cluster Resource Collector 23](#_Toc49269937)

**Revision Records**

|  |  |  |  |
| --- | --- | --- | --- |
| Date | Revision Version | Change Description | Author |
| 2020-08-25 | 1.0 | First version of Global Scheduler Design Doc | Cathy Hong Zhang |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

# Introduction

## Problem Statement

Edge Cloud brings computing resources to the edge of the network and provides secure, low latency, and high QoS computing service to mobile applications/devices at the edge and mitigates the network congestion on the Internet. Edge Cloud has several characteristics: the number of Edge Cloud Sites is large, they are all distributed, and each one has small limited resource pool. It could happen that a resource request, e.g. a VM or container request, cannot be served by an Edge Cloud site due to resource limitation. So, resource sharing and coordination among multiple Edge Cloud Sites is needed, i.e. Edge Computing must support resource scheduling in a global way.

Global scheduling is also needed in Public Cloud and Hybrid Cloud. At any given time, one cloud data center may have resource strain while another cloud data center may be in light usage. Global scheduling makes better scheduling decision and renders better resource utilization and service performance.

## Goals and Scope

Global scheduler aims to provide a service that breaks resource boundary, extends resource limitation, and optimizes resource utilization among cloud edge sites and cloud data centres. It has a global view of all the DCs’ available resource capacity and edge sites’ available resource capacity, and intelligently places VMs/Containers on a DC/edge site that meets the VM/Container specification and achieves optimal global resource utilization. The following are the main **goals** the Global Scheduler platform is aiming to achieve:

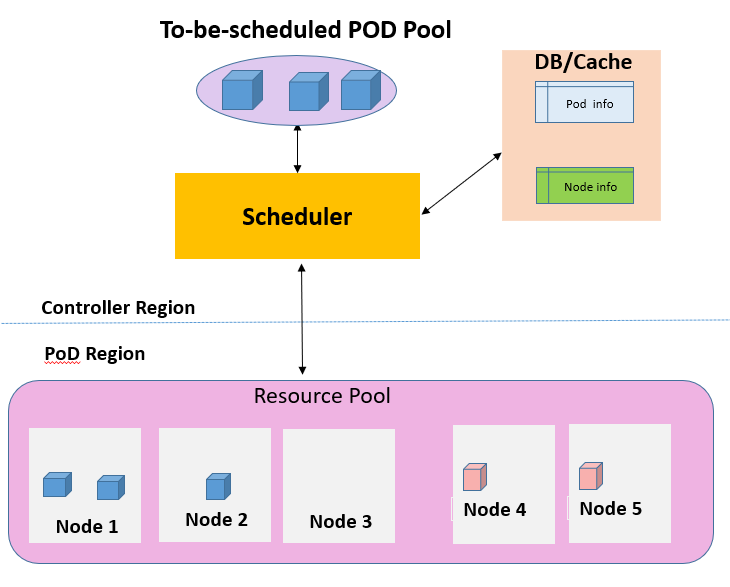
* Scalability, up to 10K DCs+Edge Sites
* Low Scheduling Latency, within one second
* High resource utilization
* Multitenancy
* Reliability/Availability
* Unified scheduling API for VM and container
* Shared resource pool for VM and container
* Locality awareness
* Flow traffic locality-based resource migration
* Flow traffic volume-based resource auto scale-out

## Design Considerations

There are multiple types of scheduler architecture design. The following sections describe each design and analyze its pros and cons

### Monolithic Scheduler (Kubernetes, Openstack)

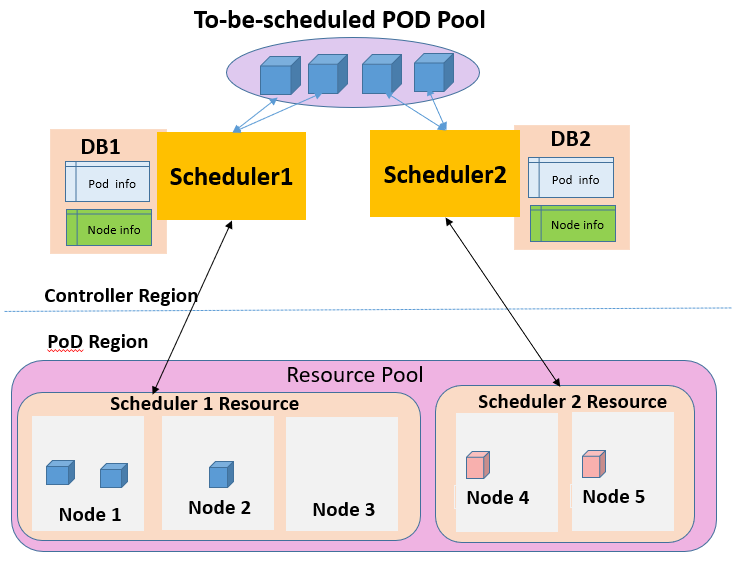
The monolithic scheduler architecture uses a single, centralized scheduler for all VMs/Containers. The single scheduler orchestrates and manages all the node resources. Current open source Kubernetes scheduler and Openstack Nova scheduler fall into this category. One big issue with this design is its scalability and throughput limitation. This design obviously does not scale up to the required global size and cannot meet the global throughput requirement.



### Siloed Multi-Scheduler

In this design, node resources are partitioned into multiple groups with each partition associated with a unique scheduler. Resource information is not shared across partitions/schedulers. Each incoming scheduling request is sent to only one scheduler. These scheduling requests are usually distributed to a scheduler in a round robin way or load balanced way.

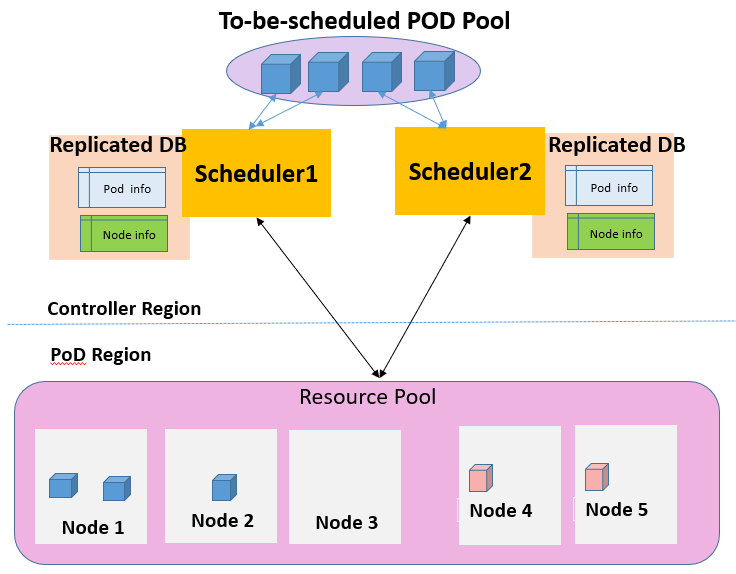
This design has good scalability and there is no need to deal with conflict/interference. But scheduling is not globally optimal and resource usage rate will not be good since node resources are not shared. Scheduling may fail for a VM/Container request if at that specific time point the node pool’s available capacity is smaller than what the VM/Container requests.



### Shared-State Multi-Scheduler

In this design, node resources are replicated and shared by all schedulers. Each incoming scheduling request is sent to only one scheduler. These scheduling requests are usually distributed to a scheduler in a round robin way or load balanced way.

This design has good scalability and scheduling is globally optimal. Resource usage rate will be good since node resources are all shared. But it needs to deal with information syncing among these replicated DBs. Another key issue is the scheduling conflict/interference when these concurrently running schedulers select the same node for their VM/container requests, which exhausts the node’s resource.

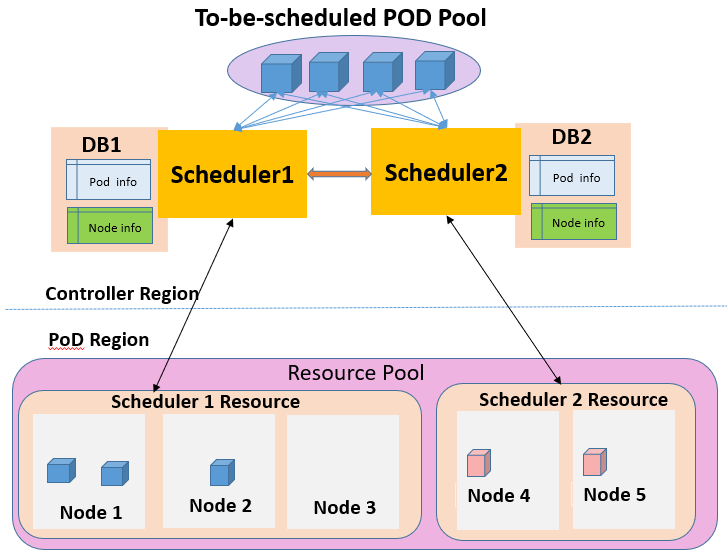


### Siloed Multi-Scheduler with Intercommunication

In this design, node resources are partitioned into multiple groups with each partition associated with a unique scheduler. Resource information is not shared across partitions/schedulers. Each incoming scheduling request is sent to ALL schedulers instead of one scheduler.

All the schedulers run their scheduling algorithms in parallel. Each scheduler selects the best node from its own resource pool according to a ranking score. They then communicate their best node’s ranking score with each other to reach consensus on the best node.

In this design, scheduling will be globally optimal and resource usage rate will be good. There is no scheduling conflicts/interference. But scalability won’t be good since each Request is essentially handled in a serial way. Allocation rate will not be good. It also introduces a lot of traffic into the control plane network. Implementation is also complicated since a) a consensus/convergence protocol needs to be implemented. b) two-phase commit needs to be implemented, i.e. each scheduler needs to reserve the resource from its selected node and then either commits the resource or releases the resource after the consensus is reached.

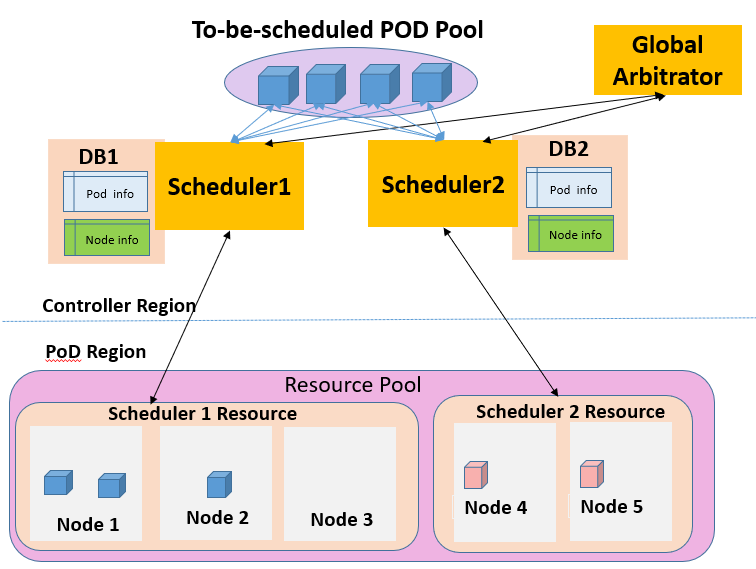


### Siloed Multi-Scheduler with Arbitrator

In this design, node resources are partitioned into multiple groups with each partition associated with a unique scheduler. Resource information is not shared across partitions/schedulers. Each incoming scheduling request is sent to ALL schedulers instead of one scheduler.

All the schedulers run their scheduling algorithms in parallel. Each scheduler selects the best node from its own resource pool according to a ranking score. They then communicate their best node’s ranking score with a common Arbitrator. The Arbitrator chooses the node with the highest score and then notifies each scheduler of the decision.

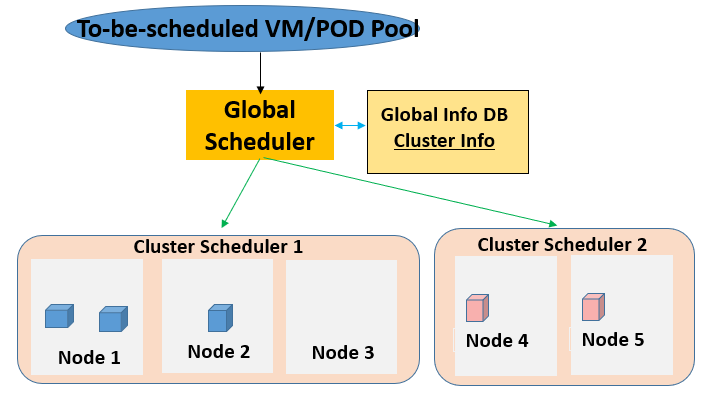
In this design, scheduling will be globally optimal and resource usage rate will be good. There is no scheduling conflict/interference. But scalability won’t be good since each Req is essentially handled in a serial way. Allocation rate will not be good either. It also introduces a lot of traffic into the control plane network. Implementation is also complicated since a) a Global Arbitrator needs to be implemented. b) two-phase commit needs to be implemented, i.e. each scheduler needs to reserve the resource from its selected node and then either commits the resource or releases the resource after receiving final decision from the Global Arbitrator.



### Two-Level Scheduler

Scheduling is divided into two levels: one first-level scheduler/manager and multiple second-level cluster schedulers. The first-level scheduler selects the best cluster and the selected cluster level scheduler selects the best node.

In this design, scheduling could be globally optimal and resource usage rate can be good. There is no scheduling conflict/interference. But scalability and allocation rate won’t be good since all the requests go through a single first-level global scheduler. The first-level global scheduler can become a bottleneck.



There is a variation to this architecture---the first-level scheduler/manager “selects” the best cluster by offering compute resources to multiple parallel, independent second-level schedulers. That is, the first-level scheduler/manager decides how many resources to offer each cluster scheduler based on some organizational policy, such as fair sharing or priority, while cluster schedulers decide whether to accept the resource and which VMs/Containers to run on them. This two-level scheduling architecture appears to provide ﬂexibility and parallelism, but in practice the scalability is limited by the capacity of the first-level scheduler/manager and the locking algorithm. The offer-reject cycle introduces extra latency, which impacts the allocate rate.

### Summary of Scheduler Design Pain Points

Global scheduling must meet several goals simultaneously:

* High scalability to support high volume of incoming scheduling requests.
* Rapid decision making and low scheduling latency to support a high allocation rate
* Globally optimal scheduling to support high resource utilization across all cloud data centers and cloud edge sites
* Satisfying the user-supplied placement constraints

These goals usually conflict with each other. As analyzed in the previous sessions, each type of scheduler architecture only meets a few of the above goals, but not all the goals.

# Global Scheduler Overall Architecture Design

## Key Requirement Analysis

Scalability is a key requirement in current scheduling architecture design. Since a large amount of resource scheduling requests come through the Global Scheduler, the Global Scheduler is at risk of becoming a scalability bottleneck. A single monolithic Global Scheduler cannot meet the scalability requirement. The architecture must have multiple Global Schedulers running in concurrency with incoming VM/Container requests load balanced to them. But concurrent execution introduces potential scheduling conflicts. Section 2.5.3, 2.5.4, and 2.5.5 will describe the mechanisms to mitigate and eliminate the conflicts.

High resource utilization is always a key requirement. Global resource allocation/scheduling is a feasible way to achieve high resource utilization. To support global optimal scheduling, resource sharing is the way to go. Section 2.4.1 will describe how resource is shared by all global schedulers.

Low latency is another key requirement in Edge Cloud. We need to support a scheduling placement that is near the end user’s geolocation. We also need to minimize the scheduling latency. If we go for one-level architecture and the scheduling work across all public cloud DCs and Edge Sites is centralized to a few Global Schedulers, the Global Schedulers’ shared database will be over bloated with a huge number of nodes’ information structures. We would also need many Global Schedulers running concurrently. All these running Global Schedulers will go through every node’s allocable capacity information structure in the shared DB simultaneously. The DB’s atomicity and consistency operation will make it hard to meet the low latency requirement. In addition, many DCs or Edge Site clusters already implemented their own sophisticated cluster level VM/Container scheduling. Eradicating all those cluster level scheduling and moving the functionality to the global level scheduler would require expensive refactoring. So, in our design, we divide the scheduling into two levels. The first level Global Scheduler selects the best cluster for the VM/Container request. We then leverage existing cluster level scheduler to select the best node that will host and run the VM/Container. Since node level scheduling is offloaded to the second level cluster scheduler and those cluster schedulers run in parallel with access to their own independent resource DB, scheduling latency is greatly reduced.

In addition to the above requirements, the global scheduling architecture must be able to integrate with any type of clusters on the southbound, such as Kubernetes Clusters, Arktos Clusters, Openstack Clusters, etc.. Priority Scheduling and Fair Scheduling should also be considered in the design.

## Key Points of this New Scheduler Architecture Design

Our solution is a new two-level concurrent scheduler architecture built around globally shared resource pool. This new design aims to achieve both high scalability, low scheduling latency, and high resource utilization while meeting the user-supplied placement constraints through a combination of the following mechanisms:

* shared state lock-free optimistic concurrent scheduling operation
* multi-dimensional optimization-based scheduling algorithm
* dynamic geolocation and resource profile-based partition scheme
* conflict aware VM/Container request distribution and scheduling mechanism
* smart conflict avoidance approach

## High Level Architecture Design

We divide the scheduling into two levels. The first level Global Scheduler selects the best cluster for the VM/Container request. We then leverage existing cluster level scheduler to select the best node that will host and run the VM/Container. Our Two-Level Scheduler Architecture is different from what is described in section 1.3.6 since our first level Global Scheduler is not a single monolithic scheduler. It employs a distributed model which consists of multiple Global Schedulers running in a lock-free concurrent mode, thus our new design avoids the issues of scalability bottleneck and low allocation rate. The architecture allows flexible scale-out to the needed global capacity.

In our design, all the Global Schedulers operate in a shared-state manner. That is, they shared one global resource DB which holds the cluster-level resource information of all the clusters. The DB size will be a manageable size since the DB only holds cluster level resource information, such as available cluster CPU, available cluster memory, supported VM flavor list, and network topology of clusters, network proximity of clusters, etc.. Unlike the Shared-State Multi-Scheduler Architecture outlined in section 1.3.3, the global resource DB is not replicated by each Global Scheduler. Instead, they share and read/update information from one global resource DB. In our new design, the Global Schedulers run in an optimistic concurrent way instead of using pessimistic concurrency control, which removes the locking latency.

Unlike existing multi-scheduler architecture design as described in section 1.3.2 and 1.3.3, in which the incoming scheduling requests are either distributed to all schedulers or to a scheduler in a round robin way, in our design, the incoming scheduling requests are distributed to a Global Scheduler based on the request’s profile. Section 2.5.4 illustrates how this distribution mechanism combined with the Global Scheduler partitioning scheme (Section 2.5.3) will mitigate the scheduling conflict when all the Global Schedulers run concurrently.

In summary, this design eliminates the issues associated with existing Two-Level Scheduler Architecture, Siloed Multi-Scheduler Architecture, and Shared-State Multi-Scheduler Architecture. It renders good scalability, low latency, and global optimal scheduling while satisfying the user-supplied placement constraints.

On the southbound, it integrates with different types of clusters via a flexible plugin design.

Figure.1 is an illustration of the high-level architecture.

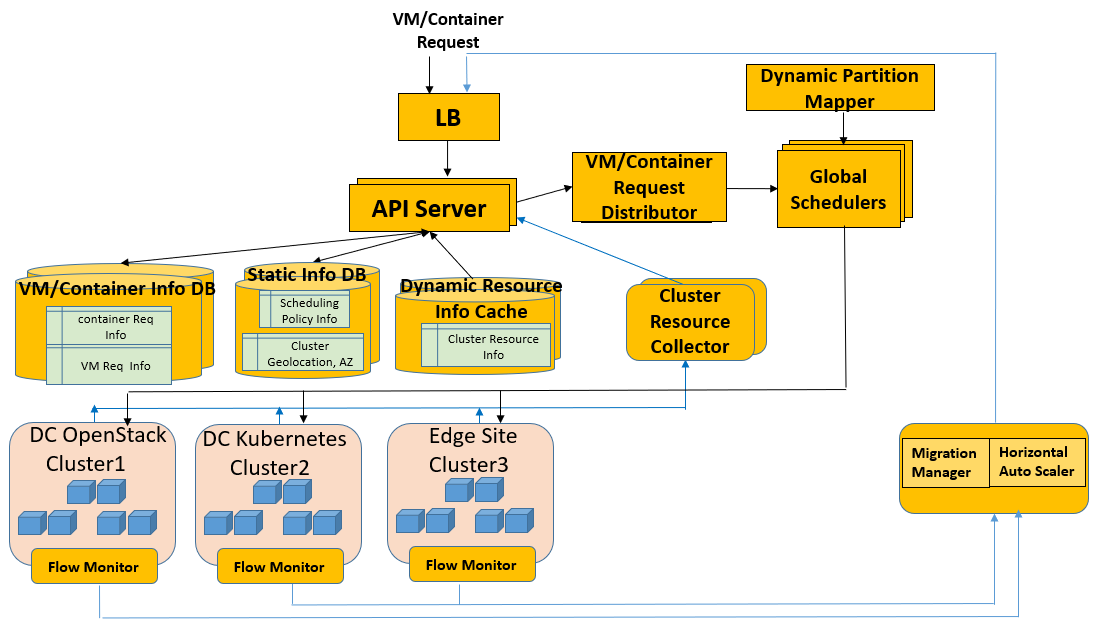


Figure.1---Global Scheduling Architecture

In the future, we may explore whether this two-level architecture may flatten into a one-level architecture.

## Architecture Component Description

The following table describes each component/subsystem in the Global Scheduling Architecture.

|  |  |  |
| --- | --- | --- |
| **Subsystem or Component** | **Description** | **Roles and Responsibilities** |
| **API Server** | API server exposes VM/Container CRUD APIs to clients, who use the Global Scheduler Service to request/read/update/delete VM/Container resource | * Receive, authenticate, authorize and validate API request and perform various admission controls. * Perform data transformation and persist VM/Container information in Database cluster * Act as a proxy between the Request Distributor and the Database cluster, e.g., watch for VM request addition to DB and inform Request Distributor of this new VM request. * Support multiple API Server partitions for scalability * The API Server design will reuse existing K8S declaration-based API server design and leverage Arktos API server scalability and multi-tenancy design/implementation. |
| **Data Store for Static VM/Container Info, Scheduling Policy Info, Cluster’s Geolocation/Region/AZ** | Distributed database to store the VM/Container request profile and states, scheduling policy, each cluster’s geolocation, region, AZ, supported VM flavor list. ETCD will be used for its consistency, persistency, and high availability. ETCD serves as the distributed, high availabe, multi-version data store with strong consistency, as well as messaging and/or eventing machanism for the system, i.e., when a key value changed, it delivers the changes to components which “watch” the change events for the key. | * Distribute data store for high availability * Support Data Persistency * Support both strong and eventual data consistency * Support multiple partitions for scalability * Notify object changes to API server which will notify the VM/Container Request Distributor component |
| **Dynamic Cluster Resource Info Cache** | Distributed database cache to store the Cluster’s available resource capacity. Since this information changes dynamically and will be collected periodically, there is no need to persist this information. Either ETCD Cache or Ignite satisfies our need for scalability, high availability, and cache performance  Each Global Scheduler partition’s workload, geolocation tag, and profile tags are also saved in this DB cache | * Distribute data store for scalability and high availability * Support cache data store for high performance * Support fast read operation * Support eventual data consistency * Notify object changes to API server which will notify the VM/Container Request Distributor |
| **VM/Container Request Distributor** | The Distributor tags each incoming VM/Container request based on its specification of geolocation/region/AZ and resource profile. It then assigns a Home Global Scheduler to it based on tag matching (each Global Scheduler is also tagged based on its assigned geolocations/regions/AZs and resource profiles supported by those geolocation/regions/AZs). If there are two or more matching Global Schedulers, the Distributor assigns a Global Scheduler that has the lowest load to it. | * Register with API server to get notification of new or changes of VM/Container object * Distribute the VM/Container scheduling requests to an appropriate Home Global Scheduler * Support multiple partitions for scalability. If multiple partitions are needed, incoming requests from the API servers will be load balanced to those Distributor partitions. That is, each Distributor will be connected to a subset of API Servers and will handle a subset of incoming requests. * Each Distributor is connected to all Global Schedulers and will distribute an incoming VM/Container request to a Home Global Scheduler based on tag matching |
| **Dynamic Partition Mapper** | The Mapper monitors the workload on each Global Scheduler and dynamically adjust each Global Scheduler’s assigned geolocations/regions/AZs and corresponding resource profile tags. | * Determine the set of geolocations/regions/AZs each Global Scheduler is associated with. * Construct the corresponding resource profile tags for each Global Scheduler * Monitor the workload of all the Global Schedulers. * Dynamically adjust the assignment of geolocations/regions/AZs and corresponding profile tags to avoid overloading any single Global Scheduler. * Update each Global Scheduler’s workload, geolocation, resource profile tags in the DB cache |
| **Global Scheduler** | The Global Scheduler runs the scheduling algorithm to select the best cluster for the incoming VM/Container Request. The operation consists of two steps. The first step is filtering, which filters out clusters which do not meet the VM/Container’s geolocation requirement, resource requirement, other constraints etc.. The first step produces a candidate list of Clusters. The second step is ranking, which applies a weighted multi-dimension optimization algorithm to score each cluster in the candidate list and select the cluster with the highest score. | * Accept scheduling requests of VM/Container submitted from the Scheduling Request Distributor. * Run the scheduling algorithm to select the best cluster * Send the VM/Container Request to the selected cluster which will schedule/select a best node to run the VM/Container and provision related network and storage resources * Update Data Store with the scheduling result. * Update the selected cluster’s allocable capacity in the Data Store * Internally maintain and manage short run queues of VMs/Containers to be scheduled as well as short run queues of scheduling decision. * Manage re-scheduling in case of scheduling failure response from the cluster scheduler * Support multiple schedulers running concurrently to meet the scalability and latency requirement |
| **Cluster Resource Collector** | Collect and Construct the cluster level resource information every few seconds and sends them to the Data Store which will cache the Info | * Collect allocable capacity information from each cluster in a timely manner * Handle the interaction with the underlying Cluster Platform, such as Openstack platform, K8S platform, Arktos platform. * Support multiple partitions with each partition handling a group of clusters * Resource Data Structure design must meet the high query/update frequency and the low latency requirement |
| **Flow Monitor** | Some VM may host applications that involve a lot of interactions with the end users. Fast response and high throughput are critical requirements for such applications. The VM that hosts such an application needs to be placed near the application’s end user traffic. This Flow Monitor monitors each VM’s application flow volume, derive the application flow’s origins. It then analyzes the information and determine whether there is a need to migrate the VM to another location or to scale out more VMs to support the application flow. | * Monitor each VM’s application flow volume * Retrieve the application flow’s origins based on their source IPs * Analyzes the application flow’s geolocation and volume information to determine whether there is a need to migrate the VM to another location or to scale out more VMs to support the application flow. * Determine the number of new VMs that need to be created to serve the application traffic flow * Send migration instruction to the Migration Manager for VM migration or send scale out/in instruction to the VM Horizontal Auto Scaler for auto scaling out/in |
| **Migration Manager** | Manages the migration of VM from one geolocation to another geolocation |  |
| **VM/Container**  **Horizontal Auto Scaler** | Manage the interaction with the API server to scale out/in VMs to meet the application traffic flow capacity |  |

### 

## Key Global Scheduler Mechanism Design

### Shared State Lock-Free Optimistic Concurrent Scheduling Operation

Our Global Scheduling Architecture is built around shared state, lock-free, optimistic concurrency control, to achieve high resource utilization rate, low latency, and high scalability.

To support the scale of global scheduling, there are multiple Global Schedulers running concurrently in our design. Incoming VM/Container Requests are evenly distributed to these Global Schedulers with each VM/Container Request being sent to only one Global Scheduler.

All cluster resource information is saved in a globally shared-by-all database. Each Global Scheduler has global view and full access to all the cluster resources in the global database and has complete freedom to lay claim to any available cluster resource. The Global Schedulers run independent of each other and each makes independent scheduling decision based on the resource information snapshot taken at the time when it accesses the global database. Once a scheduler makes a placement decision for a VM/Container request, it subtracts the VM/Container’s resource from the selected cluster’s allocable capacity and updates the cluster’s allocable capacity information in the shared global DB in an atomic mode. This update happens right before it sends the placement decision to the cluster.

Conflicts may arise when some of Global Schedulers simultaneously schedule their VMs/Containers onto the same cluster, which happens to exhaust that cluster’s allocable capacity. A pessimistic approach avoids the issue by using a lock and ensuring that a chunk of resource is only allocable by one scheduler at a time. This locking mechanism essentially serializes the operation. This serialization process introduces extra latency and limits the scalability. Our design uses an optimistic approach which supports full parallelism without any inter-locking between the Global Schedulers. To reduce the chance of multiple Global Schedulers claiming the same cluster resource simultaneously, our design uses a geolocation and resource profile-based partition scheme. Our design also uses a conflict aware VM/Container distribution mechanism as well as a conflict avoidance approach to eliminate the chance of claiming the same resource.

Our optimistic approach increases parallelism and can support high scalability and low latency performance because all the Global Schedulers operate completely in parallel and do not have to wait for each other, and there is no inter-scheduler head of line blocking.

### Multi-Dimensional Optimization Based Scheduling Algorithm

The scheduling algorithm consists of two phases. The first phase is filtering. The second phase is scoring.

Different VMs/Containers have different resource requirements and constraints. Factors that need to be considered for filtering include the following:

1. resource requirements, such as number of vCPU, memory size, number of EIPs, disk volume
2. hardware / software / policy constraints, such as type of disk, GPU
3. affinity and anti-affinity specifications, such as collocated in the same AZ as another VM/Container/Storage/DB or in a separate AZ from another VM for fault tolerant purpose
4. data locality, such as a specific geolocation or a specific AZ.

All the clusters need to be filtered according to the VM/Container’s specific requirements. This filtering phase will select a list of candidate clusters that are suitable to host/run the VM/Container. If none of the clusters are suitable, the VM/Container remains unscheduled until a Global Scheduler is able to place it.

Validating a VM/Container’s specification of hardware/software/policy constraints, affinity, anti-affinity, and geolocation/AZ locality against a cluster is straightforward. But checking whether a cluster can meet the VM/Container’s resource requirements of vCPU, memory, EIPs, and disk volume is tricky. Let’s call the latter type of check “*quantity* *check*”.

Our global scheduler does scheduling/placement of a VM/Container to the granularity of a cluster. To avoid a huge DB, we should not store every node’s allocable capacity information in the global DB. But if we only store each cluster’s accumulated allocable capacity, “*quantity* *check”* cannot be done properly. This is because even a cluster’s accumulated allocable capacity is larger than a VM/Container’s *quantity* requirement, the cluster may have no single node which has enough resource to host/run the VM/Container. An “average per node allocable capacity” is a better way. But “*quantity check*” against the “average per node allocable capacity” is not good enough either. As shown in Figure 2, the new VM requires 128M memory and Cluster1’s “average per node available memory resource” is 110M memory. Although Node 3 in Cluster 1 has 256M, which meets the VM’s memory requirement, the *“quantity check”* against Cluster 1 will fail and Cluster 1 will be incorrectly filtered out of the candidate list.

Therefore, instead of doing *“quantity check”* against a cluster’s “average per-node allocable capacity”, we should do *“quantity check”* against a cluster’s “largest available node’s resource”. But checking against only the largest available node of a cluster is not enough. This is because a VM/Container’s resource requirement spans more than one resource type. A node having the largest available vCPU may have little available memory. So, we need to collect the resource information of several largest available nodes from each cluster (a few with largest CPU, a few with largest memory, etc.). Lets’ call these nodes “potential feasible nodes”. Resource information of each cluster’s “potential feasible nodes” should be saved in the global resource database. “quantity check” should be done against a cluster’s “potential feasible nodes” to see whether there is one node which has enough vCPU, memory, EIPs, disk volume to host/run the VM/Container. If so, the cluster is added to the candidate list.

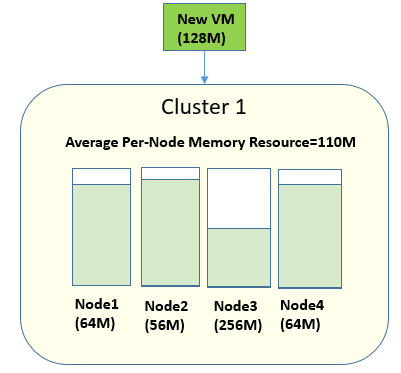


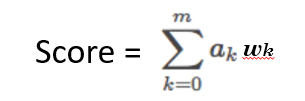
Figure.2---Node Allocable Capacity vs Cluster Average Allocable Capacity

After the scheduling algorithm finds the candidate list of clusters for a VM/Container, it goes to the second phase---scoring phase. In the scoring phase, the scheduling algorithm runs a set of functions to score each cluster in the candidate list and picks a cluster with the highest score.

Factors that need to be considered for scoring include the following (more can be added in the future):

1. average per node allocable capacity. The more allocable capacity, the higher the score. But a VM/Container’s resource requirement spans more than one resource type. A cluster may have more “per node available memory resource” but less “per node available vCPU resource” than another cluster. How to define “more” or “less”? Section 2.5.2.1 describes the way to define this.
2. resource equivalence. The more equivalent, the higher the score. The equivalence here refers to equivalence of allocable capacity among vCPU, memory, EIP, disk, etc. We do not want a cluster to be left with a lot of available vCPU but little memory after the placement of a VM/Container or vice versa. We should maintain a good balance among the various types of resources without depletion of one type of resource since if one type of resource (e.g. vCPU) is exhausted, even a cluster still has a lot of empty capacity of the other types of resources (e.g. memory), no VM/Container can be scheduled onto that cluster. Maintaining resource equivalence will enhance a cluster’s resource utilization rate and performance. What metric is used to determine an equivalence score? Section 2.5.2.2 describes the way to determine this.
3. cluster health. The fewer runtime errors, the higher the score
4. energy efficiency: the higher energy efficiency, the higher the score

The score of each factor is normalized into the range of 0-1. 0 is the lowest score and 1 is the highest. Then a weight is assigned to each factor score. The weight assignment is configurable to allow different scheduling policy. The final cluster’s ranking is calculated as a weighted inner product of these factor scores.



#### Average Per Node Allocable capacity

A cluster node’s resource spans multiple types, e.g. memory, vCPU, disk, network interfaces. Suppose there are n types of resources. We represent each node’s allocable capacity as a point in a n-dimension Euclidean space. We then calculate the centroid of these points since the concept of centroid is the multivariate equivalent of the mean. In other words, the centroid is the mean position of all the points in all the coordinate directions. The centroid is a good representation of the average per node allocable capacity of the cluster. Figure 3 is an illustration of the centroid concept in a 3-dimensional space.

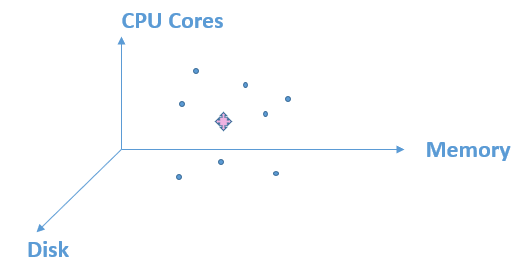


Figure.3---Centroid in a Three-Dimensional Space

Then we calculate the score for a cluster based on the VM/Container’s requested capacity to the cluster’s centroid capacity ratio. The following equation illustrates how to get the score for a cluster.

Here represents the VM/Container’s requested capacity of a resource type *k*, e.g. number of vCPU the VM/Container needs, represents the cluster’s centroid capacity (average per node allocable capacity) of resource type *k*.

#### Resource Equivalence

Workload scheduling/placement usually considers metrics like resource availability, affinity/anti-affinity, etc. If the scheduling algorithm is not designed well, it may happen that a cluster’s leftover resource is badly skewed. As shown in the diagram, there is a lot of leftover memory, but there is little CPU. This will lead to low resource utilization and poor performance.

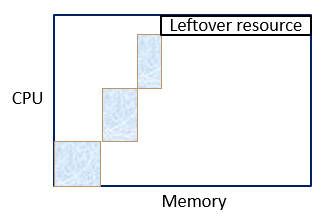


Figure.4---Resource Skew

To avoid this problem, the scheduling algorithm should take resource equivalence into consideration and try to ensure that every scheduling decision results in a good balance among the various types of resources. Let’s define some terminology:

* Empty Cluster Resource Ratio: this refers to the resource ratio of a cluster when it has no workload.
* Current Cluster Resource Ratio: this refers to the resource ratio of a cluster before it allocates resource to the new VM/Container.
* Leftover Cluster Resource Ratio: this refers to the resource ratio of a cluster after it allocates resource to the new VM/Container.

The goal is to place the new VM/Container onto a cluster so as to make the Leftover Cluster Resource Ratio matches better to the original Empty Cluster Resource Ratio. “cosine similarity” can be used to calculate the deviation between two resource ratios.

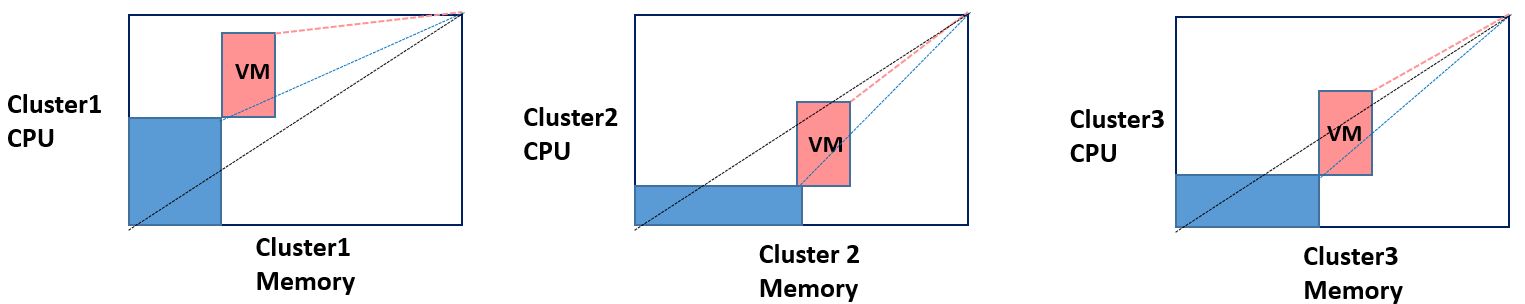


Figure.5---Cluster’s Resource Ratio Change After a New VM Placement

Placement of a new VM onto a cluster can make a cluster’s Leftover Cluster Resource Ratio move either farther from its Empty Cluster Resource Ratio or closer to its Empty Cluster Resource Ratio. As shown in the above diagram, the three clusters have the same Empty Cluster Resource Ratio. The blue box represents resource already allocated to VMs/Containers running on each cluster (The scheduling decision of those VMs/Containers may be determined by the Geolocation/AZ requirement of those VMs/Containers). When the new pink VM request comes, we can see that scheduling it onto cluster1 will make cluster1’s Leftover Cluster Resource Ratio deviates more from its Empty Cluster Resource Ratio. Scheduling it onto cluster2 will make cluster2’s Leftover Cluster Resource Ratio move closer to its Empty Cluster Resource Ratio. Same for cluster3.

The algorithm will loop through every cluster to calculate its Leftover Cluster Resource Ratio. There are two cases:

1. Placement of the new VM will make some or all clusters’ Leftover Cluster Resource Ratio move closer to its Empty Cluster Resource Ratio. For this case, the algorithm should give a higher score to a cluster whose Current Cluster Resource Ratio deviates the most from the cluster’s Empty Cluster Resource Ratio so as to bring the worst balanced cluster to a more equivalent resource status. The goal is to achieve an overall well-balanced resource ratio among all the clusters.
2. Placement of the new VM will make every cluster’s Leftover Cluster Resource Ratio move farther from its Empty Cluster Resource Ratio. For this case, the algorithm should give a higher score to a cluster whose Leftover Cluster Resource Ratio deviates the least from the cluster’s Empty Cluster Resource Ratio so as to mitigate the overall resource skew among all the clusters.

### Dynamic Geolocation and Resource Profile Based Partition Scheme

To meet the scalability requirement, we have multiple schedulers run concurrently and some of these schedulers may attempt to claim the same resource simultaneously. The partition architecture described in section 1.3.2 avoids this conflict issue but in that architecture, each scheduler has a rigid boundary and has restricted visibility of resources in the global scheduling framework, which defeats the purpose of global scheduling.

In our architecture, each scheduler has global view of resources and runs its scheduling algorithm based on global view of resources. How does it work?

As we know, a good portion of VM/Container requests specify geolocation/AZ for the VM/Container due to its requirement of affinity, anti-affinity, proximity to end user position, etc. This is especially true in Edge Cloud in which a key scheduling requirement is to schedule a VM/Container to an AZ/Cluster that is closer to the geolocation of its end user traffic.

So, in our architecture design, each geolocation/AZ is assigned a Home Global Scheduler. Each Global Scheduler can be the Home Global Scheduler for multiple geolocations/AZs and is responsible for placement of VMs/Containers onto those geolocations/AZs. But each geolocation/AZ will only have one home Global Scheduler. Each Global Scheduler will be tagged with the geolocations/AZs it is associated with. Since different geolocations/AZs may support different VM/Container resource Profiles, each Global Scheduler is also tagged with the resource profiles it is associated with. A resource profile is composed of information such as flavor, GPU, storage type (SSD, SAS, SATA), resource ratio, cost etc. Figure.6 is an illustration of this design.

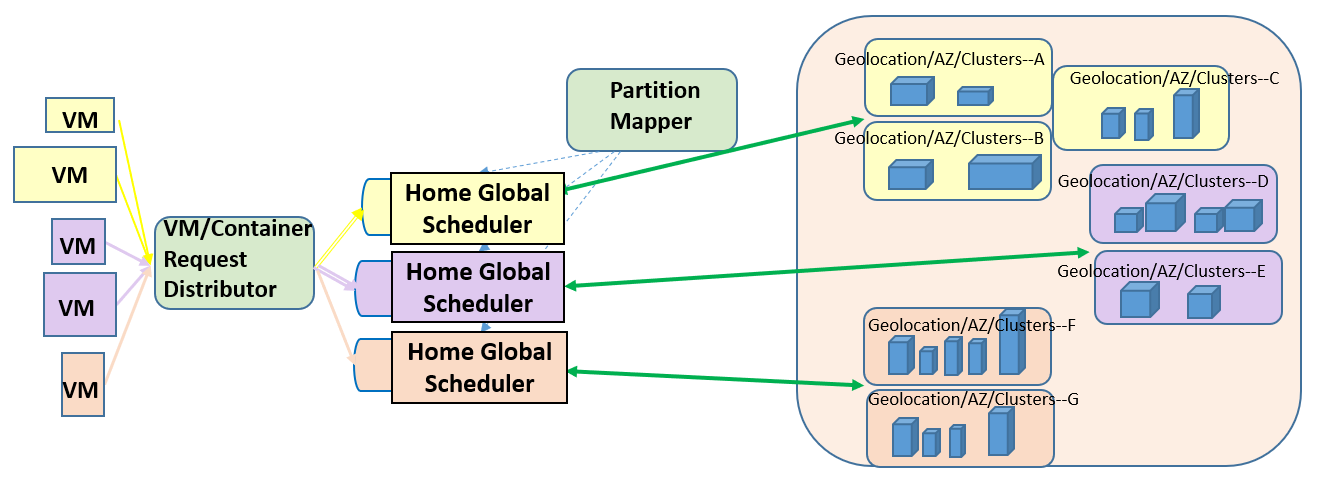


Figure.6---Global Scheduler Partition

Section 2.5.4 will describe how incoming VMs/Containers, whether or not they specify geolocation/AZ, will be distributed to these Global Schedulers.

To avoid a Global Scheduler being overloaded with incoming VM/Container requests, each Global Scheduler’s boundary can be dynamically adjusted based on load status. That is, our architecture has a Partition Mapper that monitors the load status of each Global Scheduler and will dynamically adjust the association of a Global Scheduler with the geolocations/AZs.

### Conflict-Aware VM/Container Request Distribution and Scheduling Mechanism

As shown in Figure.6, each incoming VM/Container request will be assigned a Home Global Scheduler. There are two cases that need to be handled for the assignment.

1. The VM/Container specifies a geolocation/AZ in its request. In this case, a Global Scheduler with this geolocation/AZ tag will be assigned to be the Home Global Scheduler for this VM/Container, and the VM/Container will be distributed to this Home Global Scheduler. The Home Global Scheduler will run the scheduling algorithm and select the best cluster in the geolocation/AZ to host/run this VM/Container. It may happen that there is no resource available in the specified geolocation/AZ, i.e. the scheduling algorithm generates an empty cluster candidate list in the filtering phase. When this happens, the Home Global Scheduler will run the scheduling algorithm on all the other clusters in the global framework and select the best cluster. The Home Global Scheduler can do this because it has global view of all the clusters in the global scheduling framework. But the selected cluster is not associated with this Home Global Scheduler and if the selected cluster’s own Home Global Scheduler happens to schedule some VMs/Containers onto the same cluster simultaneously and this cluster happens to be near its resource limit, scheduling conflict could arise. Section 2.5.5 will describe how to address this problem.
2. The VM/Container does not specify a geolocation/AZ. In this case, a Global Scheduler tagged with a resource profile that meets the VM/Container’s requirement will be selected as its Home Global Scheduler, i.e., VM/Container will be distributed based on match of resource profile. It may happen that multiple Global Schedulers support the same type of resource profile. In this case, the VM/Container request will be distributed to a Global Scheduler with the smallest load. The assigned Global Scheduler will run the scheduling algorithm on all the clusters in the global framework and select the best cluster. This selected cluster’s Home Global Scheduler may not be this Global Scheduler. Similarly, if the selected cluster’s own Home Global Scheduler happens to schedule some VMs/Containers onto the same cluster simultaneously and this cluster happens to be near its resource limit, scheduling conflict could arise. Section 2.5.5 will describe how to address this problem.

### Smart Conflict Avoidance Approach

As described in Section 2.5.4, the scheduling algorithm may schedule a VM/Container request to a cluster whose Home Global Scheduler is not the Global Scheduler that makes this scheduling decision. To avoid potential scheduling conflict, the Global Scheduler, which makes this scheduling decision for the VM/Container, will forward the VM/Container request to the selected cluster’s associated Home Global Scheduler for final resource check and placement. The selected cluster’s Home Global Scheduler will place this VM/Container request at the start of its scheduling queue instead of at the end of its scheduling queue. Since scheduling latency mostly comes from the wait time in the scheduling queue, placing this VM/Container request at the start of the scheduling queue ensures that the latency introduced by this “forward” is just one more cycle of scheduling algorithm calculation, which is very small compared with the “one second” scheduling latency requirement.

If the latency introduced by this “forward” is not neglectable, the following alternative way can be used to address it.

The originally assigned Home Global Scheduler for the VM/Container request will always run the scheduling algorithm among all clusters in the global framework and select two best clusters. One from the clusters associated with this Home Global Scheduler, the other from those clusters not associated with this Home Global Scheduler. If there is not a big gap among the ranking scores of the two best clusters, the cluster associated with the Home Global Scheduler will be selected and the “forward” operation is skipped. Only when the gap exceeds a pre-defined threshold, the “forward” operation will be triggered.

### Scheduling Failure

In our architecture design, we have a cluster Resource Collector which collects allocable capacity information from each cluster in a timely manner and save the cluster’s resource capacity information in the Global Scheduler’s DB. There is always a real-time gap between the resource capacity information in the Global Scheduler’s replicated DB and the actual resource capacity in the cluster DB. Also, some VM/Container requests may go to the Cluster Scheduler directly and do not go through the Global Scheduling Framework. Therefore, scheduling failure may happen due to out-of-sync resource information. No matter how frequent the Global Scheduling Framework does the resource information collection and synchronization, it is hard to guarantee 100% consistency among the two DBs unless we use a locking and strong consistency scheme. A strong consistency scheme is complicated, needs coordination and refactoring of the cluster-level DB design, and will introduce high latency. A more realistic approach is to do rescheduling if the cluster scheduler returns scheduling failure.

To mitigate the scheduling latency due to scheduling failure, our Global Scheduler will insert a failed VM/Container scheduling request to the start of its scheduling queue. Furthermore, this failed cluster will be removed from the VM/Container’s cluster candidate list

### Scheduling of Scale Set

### Priority Scheduling and Fair Scheduling to Avoid Security Attack

* Latency sensitivity, Price, customer importance, can be different for different requests.
* Fair scheduling to avoid starvation of low priority VM/container request
* Fair scheduling to mitigate hacker’s DOS attacks by limiting any single tenant’s VM/Container requests

# System Flow and Interface Design

In this section, we will describe the key system flows to further understand the platform architecture design and communication among components/subsystems.

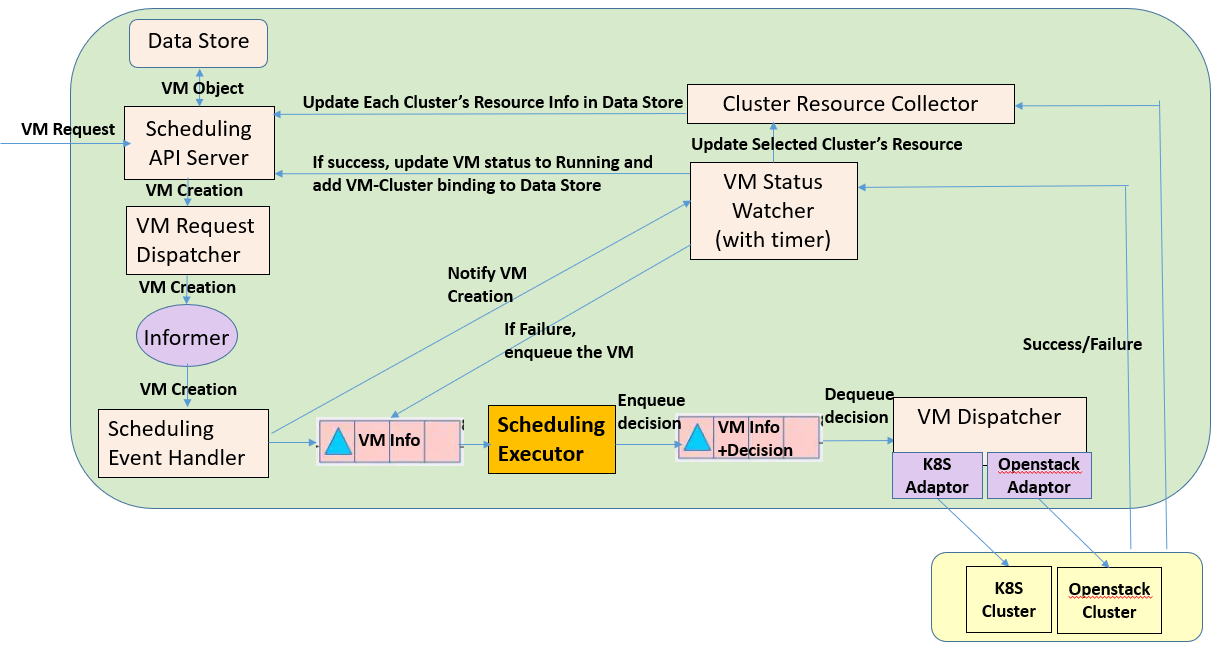


Figure.7---Global Scheduler System Flow

# Global Scheduler Subsystem Design

## API Server

### VM/Container Resource Model

### VM/Container Resource API

some requires GPU, some requires being collocated in the same AZ as another VM or Container or Storage, some requires separate AZ from another VM for fault tolerant purpose, some requires a specific geolocation

allow migration, allow auto scale out

### Scale set VM/Container Resource API

## Data Store

Total available Cluster Memory

Total available Cluster CPU

Total available Cluster Disk

Total available number of EIP

Largest node’s available CPU and Memory

Supported Flavor list

Cluster Geolocation

Cluster Region and AZ

Cluster Operator Name

Cluster’s supported Storage types

## 

## VM/Container Request Distributor

## Dynamic Partition Mapper

## Scheduling Algorithm Executor

## Cluster Resource Collector

We will put a resource agent in each cluster which reads from each cluster’s resource DB and reports the allocable capacity to Global Scheduler’s resource collector