[Design] Generalized and unified scheduling-strategies in YARN

By *Vinod Kumar Vavilapalli & Wangda Tan* with inputs from *Jian He & Varun Vasudev*

Table of Contents

(1) Introduction

Apache Hadoop YARN's *ResourceRequest* mechanism is the core part of the YARN's scheduling API for applications to use. The *ResourceRequest* mechanism is a powerful API for applications (specifically ApplicationMasters) to indicate to YARN what size of containers are needed, and where in the cluster etc.

However a host of new feature requirements are making the API increasingly more and more complex and difficult to understand by users and making it very complicated to implement within the code-base.

This document is our initial take on generalizing scheduling strategies in YARN so that we not only simplify our existing API model and streamline our implementation, but also make it extensible enough to accommodate any future scheduling-functionality.

> The following sections contain lots of information about the motivation and all the gory details of the new framework for generalized scheduling strategies. If you want to quickly check how scheduling strategies APIs look like, you can directly jump to examples under the Case Studies section.

We now first start with a brief summary of existing state of *ResourceRequests*, enjoy the long read of 30 odd pages :)

(2) Existing characteristics of ResourceRequests

ResourceRequests in YARN are a way for applications to specify their scheduling requirements. Today, through each *ResourceRequest*, applications can specify

- **Resource size (S)**
  - Indicates memory size (physical) in MB, and CPU in vcores needed
  - This as it exists today is not extensible to more resource-types like disk, network etc. An ongoing effort aims to address the extensibility and at the same time simplifying it: YARN-3926 Extend the YARN resource model for easier resource-type management and profiles.

- **Priority (P)** of *ResourceRequest*
  - Determines the relative priority of *this* ResourceRequest compared to other *ResourceRequests* within the **same** application.
  - *ResourceRequests* with lower valued priorities are served first.

- **Number of Containers (N)** needed of the above resource-size S and priority P

- **ResourceName** determining **locality** requirements

    - A string determining where the N containers above should be placed in the cluster
    - Even though it is a generic string, today YARN's ResourceManager in reality only accepts a **machine** or **rack** name where this request should be assigned to

- **Locality strength**

    - Determined by a *relaxLocality* flag - specifies whether the requirement for locality needs to be either absolutely met or can be downgraded after a while

    - There are two types of locality-strengths - hard / soft.
        - Hard locality forces the scheduler to only give resources on the specified resource-names and makes the application wait on that scheduling-condition for-ever
        - Soft locality dictates the scheduler to give resources on the specified resource-names but gives up on that preference after a server-side timeout (time measured in scheduling cycles today).

    - *ResourceName* and *Locality strength* together give YARN applications an ability to specify physical locality requirements. By physical locality, we mean affinity towards constructs like nodes, racks for which applications **need** to understand the physical layout of a cluster.

- **Label expressions**

    - Specifies the node-label expression to be used to place these container.

    - A node-label is an attribute set on a node by the administrators (through centralized/distributed configuration mechanisms) and can be used to specify more dynamic scheduling-affinity requirements compared to locality where physical topology is needed to be understood by the applications.

    - There are many possible types of node-labels, though the platform only supports the type partition today. A partition node-label can be assigned by admins to a subset of the cluster. Every node can be assigned to only one partition, unassigned nodes belong to a default partition. Queues can be configured to further share the resources within a partition (similar to queue-sharing of cluster resources).

    - Today, YARN only supports specifying a single node-label in the node-label expressions as the platform only support the ***partitions*** node-label type.

    - The idea was to support, in the future, arbitrary expressions with AND / NOT / OR semantics to support other features like *constraints* with strengths - hard / soft (YARN-3409)

- General notes
    - One of the key design choices of the scheduling API originally was to keep it as compact as possible given the use-cases at the time (MapReduce). By inverting a task-centric scheduling requirement to instead be a resource-centric requirement, there is a tradeoff of information-loss to object compactness. The total amount of state store per-application on the ResourceManager is limited in this layout, and at the same time, the total size of all *ResourceRequests* per application that can flow from the application to the RM are bounded by the cluster-size.

To summarize, existing *ResourceRequests* essentially are grouped by the request-priority and requested resource-size, and scheduler picks machines according to other attributes like requested locality such as host/rack, label-expression etc.

Note that there are related objects like *ResourceBlacklistRequest* (list of strings representing *ResourceNames* to avoid while placing containers) and *ReservationRequest* (used by client to submit new reservations for future allocations to YARN) which serve related functionalities but in extra dimensions. In the rest of our discussion below, we refer to *ResourceRequests* as an umbrella term for all sort of scheduling requirements, attempting to unify these disparate concepts.

(3) Different types of newer scheduling requirements

We now do a survey of different types of scheduling requirements that we have already seen from various workloads running on YARN. We do this to establish a guiding compass and to avoid the pitfall of creating something as an intellectual exercise and instead focus on developing the framework with a goal of addressing concrete use-cases in the wild.

Note that the terminology used below describing scheduling-features is simply a reflection of how certain things have been called so far in the YARN community and may not always best-represent the type of scheduling requirement we are talking about. Later in our new proposal, we propose a consistent, well-defined terminology with the new framework in mind.

Locality

Locality is a mechanism introduced mainly for data-processing applications to be able to exploit closeness to the data in a cluster. There are two types of locality specifications possible today.

- Soft locality:
  - "Take me to to ***this machine(s)*** or ***this rack(s)*** or ***this machine(s) but falling back to this rack(s) after waiting for a while"***
  - This is largely a feature inherited from legacy MapReduce-only scheduling from Hadoop-1 and further used in frameworks like Apache Tez
  - YARN-80 originally re-added some of the delay-scheduling semantics from Hadoop-1 to YARN
  - **Delay scheduling**: Our existing delay scheduling only works for host/rack and it's a global setting for all applications determining how many node heartbeats are needed to pass before we relax the requirements. YARN-4189 was filed to improve wait mechanism but only for the host to rack schedule.

- Hard locality aka Whitelisting:
  - "Only take me to ***this machine(s)*** or ***this rack(s), I'm willing to wait forever to get an allocation there"***
  - YARN-392 and YARN-398 originally added these semantics to YARN.

Anti-locality

Anti-locality is a way of requesting YARN to **not** allocate resources in specific parts of a cluster. We call this blacklisting in our code-base. This is again of two types:

- Hard anti-locality:
  - "Do NOT take me to to ***this machine(s)*** or ***this rack(s)"***
  - YARN-392 and YARN-398 originally added these semantics to YARN along with whitelisting

- Soft anti-locality:
  - "Try not to take me to ***this machine(s) or this rack(s), but if you can't find anything else, it's okay to fall back to this machine/rack."***
  - Today this notion is ***not*** implemented in YARN.

Node labels: Partitions and Constraints

Through node-labels, we implemented in YARN a way for administrators to specify labels for nodes with the labels representing characteristics like OS, processor architecture, presence/absence of GPUs etc.

- The type of node-labels supported today are *partitions.* YARN-796 added this functionality

    - Partitions are attributes on nodes (configured either centrally or in a distributed fashion) that also have an implication of capacity sharing. For e.g, in CapacityScheduler, one can assign partitions accessible from any application of a specific queue.
    - Applications can specify labels on resource-requests - determining the partition of the cluster where they wish to run.
    - For queues with a default label expression, apps do not need to specify this explicitly.

- YARN-3409 proposed a notion of more generic labels on node that can be treated as soft / hard constraints on scheduling.

Overall, from applications' perspective, node-labels boil down to requests of type "Take me to a **machine** or **rack** that"

- Belongs to a ***specific partition (node-label)***
- Has a ***specific node-attribute(s) (constraint)***
- With each of the above be either hard or soft requirement(s)

Affinity

Affinity as introduced at YARN-1042 is a more general way of specifying likeness towards places outside of the context of data-locality.

From applications' perspective, affinity requests are of type: *Take me to a **machine** or **rack** that satisfies **a condition***. Examples include

- Machines where any of my previous containers within the same application are running
- Machines where any of my previous containers of a specific *type* within the same application are running (*take me to my already running mappers)*
- Machine where one of my containers with a specific *ContainerID* within the same application is running
- A place that satisfies any of the above conditions across multiple applications (application/service affinity)
- With each of the above possibly be either hard or soft requirement(s)

This isn't implemented yet in YARN.

Anti-affinity

Introduced together with affinity at YARN-1042, this is a general way of specifying disinclination towards specific resources in the cluster.

The requests can be of type: *Take me to a machine or rack that satisfies **a condition.*** Examples include

- Machines where none of my previous containers within the same application are running
- Machines where none of my previous containers of a specific type within the same application are running
- Machines where one of my specific containers with ContainerID within the same application is NOT running
- A place that satisfies any of the above conditions across multiple applications (application/service anti-affinity)
- With each of the above possibly be either hard or soft requirement(s)

This isn't implemented yet in YARN.

Gang scheduling

YARN-624 introduced scheduler enabling the running of a set of containers when they can all be run at the same time. This is usually a requirement when real progress can only be made when a minimal number of containers are all up and running at the same time.

This isn't implemented yet in YARN.

Combinations of above placement requirements

Some or all of the above placement requirements can potentially be mixed to use together. For example:

- Gang scheduling + anti-affinity : *give me 10 containers together and none of them should be on the same node/rack.*
- Delay scheduling + node partition: *give me 10 containers on hosts from partition X, if I cannot get them within 5 mins, any hosts are fine.*

Without a proper model, most of these features that need new work in YARN will stretch the current code and API design to far beyond a breaking point of maintainability and comprehension. Even if we somehow accomplish that, either adding newer features beyond these or figuring out the mixing/matching of different requirements is very likely an intractable exercise.

(4) A proposal for "Grand Unified Theory of Scheduling (GUTS) strategies" in YARN

Below, we take a crack at developing a framework for extensible scheduling.

(4.0) Back to basics: *"There is a place and time for everything"*

There are a lots of different aspects to scheduling in YARN. Going back to basics, scheduling facilitates *resource-requestors* to get resources provided by *resource-providers*. This exercise is defined by questions on different dimensions.

- ***Who*** is supposed to get these resources?
    - In YARN, we have apps, users, queues on etc demand side and machines on the supply side.
- ***Where*** am I supposed to place this allocation? Nodes, racks, close to other applications?
- ***How*** should the allocation be done? All or nothing? Trickle down allocations in batches as and when they are available?
- ***When*** should I return allocations? Right-away? After a specific point in time?

Besides these fundamental questions, there are a lots of different aspects to scheduling in YARN - be it sharing models via queues, limits on resource-usage in a multi-tenant cluster, access control for different users/ placement-sets etc. In the remainder of this document, we only focus ourselves on the *where*, *when* and *how* questions.

(4.1) Overview of various Objects

We start developing the framework - and thus a new scheduling API we call ***the GUTS API*** - with an overview of various objects involved in the request specification:

- ResourceRequestCollection = List of ResourceRequest objects ordered by priority
- ResourceRequest is the unit of allocation with the following fields
    - Allocation ID
    - Meta information
        - Priority
        - Allocation Tags
    - Quantity Conditions
        - Maximum-concurrency
        - Minimum-concurrency
        - Resource size of each allocation
        - Maximum number of allocations
    - Placement Strategy
        - Defines where and how much resources we want. Support nest expressions for complex placement requirements. (Details see below)
    - Time conditions

(4.2) ResourceRequestCollection

Applications can submit *ResourceRequestCollections* over time to YARN.

ResourceRequestCollection is the top-level API object and is simply a list of individual ResourceRequest objects, all of which are independent of each other and each carrying their own *Allocation-ID*. The only relationship that scheduler attributes across different ResourceRequest objects today is that they are ordered by the priority field.

(4.3) ResourceRequest

Applications can use *ResourceRequest* objects to ask scheduler for specific allocations. It has further sub-fields as described below

(4.3.1) Allocation-ID

Allocation-ID is a ***mandatory*** unique identifier for different ResourceRequests from the same application.

Allocation ID is thus a part of every ResourceRequest. The application should simply generate a unique identifier within the application - if not the client-libraries can do so. Scheduler also returns the allocation-ID as part of the response so as to help applications map the responses to the original requests.

Allocation-ID is also the unit of the scheduling API: Future requests mapped to an existing Allocation-ID will be treated by the scheduler as commands to edit/modify/augment the original *ResourceRequest*.

We will discuss more details on how this can be used in the following sections.

(4.3.2) Meta information

Meta information is used by scheduler to group resource allocations - it has the following fields

Priority

Priority is a ***mandatory*** field used to determine the scheduling order of multiple ResourceRequests from the same application.

Lower priority ResourceRequest can get resources only if no more pending resources are required from ResourceRequests of higher priority.

Allocation-Tags

Allocation-tags is an option field usable by the scheduler and app developers to group allocations:

- Every allocation can be associated with a limited number of allocation-tags. Each allocation-tag is a key-value pair of strings.

- Applications can use an allocation-tag to group and organize pending and satisfied allocations. For example, a long-running service can assign different role-types as allocation-tags to different allocations, so that it can easily understand how much resources are assigned to different roles.

- Applications can also use *target-allocation-tags* as part of a placement-strategy described in the later sections. When an application refers to a target-allocation-tag in the scheduling request, scheduler will use it to allocate resources for a ResourceRequest respecting affinity/anti-affinity to another ResourceRequest identified by the passed-in allocation-tags. We will discuss more details about this in section 4.4

Allocation-tags are scoped under an application (application-id). So, if *app A* wants to refer to *app B*'s allocation-tags as target for placement-strategy (described later), *app A* must specify both the application-id of *app B* as well as the target tags. In the future, we can see if there is a need to expand this scoping control to be user-name, cluster-id etc.

(4.3.2) Quantity Conditions

This part of the *ResourceRequest* identifies the "*how to schedule*" part of our story and answers questions of the kind *how big the allocation should be*, and *in what order the allocations should be returned* etc.

Resource size of each allocation

Indicates an extensible vector specifying the memory size (physical) in MB, CPU in vcores etc needed as part of the allocation.

Applications ***must*** specify the size of resources needed for each allocation through this field.

Maximum number of allocations

Applications ***must*** specify how many resources overall are needed for each scheduling-request through this field. This determines the overall limit on the total number of allocations that the scheduler ever returns as response to this *ResourceRequest.*

In general, most applications just set this field *only* once for the entire *ResourceRequest* - maximum-number-of-all-allocations within a *ResourceRequest* typically doesn't change per placement-strategy (defined later in the document). In cases, where applications have more complex placement requirements of the type "*give me 10 allocations on this machine or 20 allocations on that rack*", they must still mention a *ResourceRequest*-level maximum-number-of-allocations but can choose to augment the request with num-allocations per-placement-set and specify it as part of the placement-strategy.

Minimum-Concurrency

Minimum-Concurrency determines the answer to the quantity-related scheduling question "*must I return all the allocations in one shot after (server-side) pooling for a while or can I return the them as and when they become available?"*

Minimum-concurrency is an ***optional*** value that determines the lowest size of the allocation set needed by the application to make progress. Obviously, scheduler can return allocation-sets bigger than this value. It can be used to support use-cases like gang-scheduling. Scheduler will pool allocations and not return them to the application till the number of allocations reaches the value specified by minimum-concurrency.

Minimum-concurrency can take as value a single positive integer - for e.g. *minimum-concurrency = 5*. The lowest possible value is 1, essentially one allocation at a time. The default value is also 1.

An example specification of minimum-concurrency is a *ZooKeeper* service running on top of YARN that will work only if it has 2 or more daemons running concurrently.

Please note that, minimum-concurrency is only guaranteed at the first response send back to the application for this allocation. In the above ZK example, it is guaranteed that the application receives at least 2 allocations at first, all further allocations can come in without any guarantees of concurrency. Further, it is also possible that the application releases allocations so that the concurrent number of running allocations can drop to less than 2 - there's no guarantee that scheduler can **immediately** allocate resource when AM releases containers to make sure minimum-concurrency will be guaranteed.

A side note: The reservations feature proposed at YARN-1051 can pave a great way for implementing minimum-concurrency. A gang schedule is easily understandable by users and easily implementable too if it comes along with a reservation that is created upfront. If there are no upfront reservations, depending on the state of the cluster's demand and supply, it may or

may not be possible to satisfy the allocation at all! So, we do think minimum-concurrency works very well when done together with reservations, but overall, our current proposal is a more general form that is decoupled from reservations and applicable at per-ResourceRequest level.

Maximum-Concurrency

This answers the scheduling question "*should I return the allocations batch by batch, restricting the total batch size at any point of time*".

Maximum-concurrency (aka *the width)* is an ***optional*** value that limits the total number of outstanding allocations for a *ResourceRequest* at any single point of time and can be used to avoid a single *ResourceRequest* taking over too many resources in a queue.

Maximum-concurrency can take as values a single positive integer - for e.g. maximum-concurrency = 100. The lowest possible value is 1, essentially one allocation at a time, and the default value is the maximum number of allocations.

MAPREDUCE-5583 enabled the same feature but only in the MapReduce land and applicable at the entire job level (as opposed to per-allocation).

Where to specify quantity-conditions

We have made a conscious choice to place everything of *quantity-conditions* except maximum-number-of-allocations only at the top-level in the ResourceRequests. If users have complex requirements where they have to mix/match place, size etc with time within a single ResourceRequest, they should visit the section API design choices and workarounds.

Examples of Quantity conditions

In the example below, an application requests the following

- 50 total allocations
- Resource size of each application is 2G
- Minimum-concurrency is 4 meaning the application expects to have at least 4 allocations returned as part of the first allocation response for the request 12345.
- Maximum-concurrency is 8 meaning the application cannot have more than 8 allocations running at the same time under the allocation-id 12345.

```
"12345": { // Allocation_id
  // Other fields..

  // Quantity conditions
  allocation_size: 2G,
```

```
  maximum_allocations: 50,
  minimum_concurrency: 4,
  maximum_concurrency: 8,
}
```

(4.3.3) Placement Strategy

The *placement-strategy* part of the *ResourceRequest* identifies the “where to schedule” part of our story and answers questions of the type - *on which machine should the the allocation should be*, or *which part of the cluster should not be considered for allocation* etc. Default placement strategy is that you can allocate resources on any hosts.

We now describe important concepts that help application-developers specify their placement requirements.

Placement-Sets

Scheduler can allocate resources on *placement-sets*. A *placement-set* can be

- A single-element set with only one *machine* satisfying a condition or
- A multi-element set with a group of machines satisfying a condition

A *placement-set* has three properties

- A ***condition*** that defines the elements of the set
- A ***placement set type*** that determines the type of elements in the set. Note that all sets eventually get translated to machines by the scheduler for assignment decisions. Examples of set-types: *Machine* (the default type), *rack*, *partition*.
- And an optional ***maximum number of allocations*** needed in this placement-set. If this is not specified, scheduler simply limits the number of allocations based on the top level *maximum_number_allocations* in each ResourceRequest.
- An optional ***delay_to_next*** can be specified as an additional attribute on the placement-set. The delay acts ***between*** two *ordered* placement-sets. Underneath, it can either be measured as walltime or missed scheduling opportunities. It determines the amount of time scheduler waits before moving on to the **next** placement-set.
    - It is only respected if the parent-operator is *ORDERED OR (See below).*
    - For example, a machine learning application can use GPU to accelerate its execution, but it can run CPU as well. In this case, application will be able to say “*I prefer to run on GPU nodes, Non-GPU nodes are also accepted after 10 min*”.

An example of a placement-set is *a set of nodes that belong to rack /r1* ***and*** *partition p1*.

Operators acting on Placement-Sets

One can mix and match a bunch of ***disjoint*** *placement-sets* together by making use of well-known set-operators.

- ***Null operator***:
  - A single placement-set
  - E.g:
    - *Place me on "this host"*
    - *Place me on "that rack"*

- ***Negation operator***
  - Negation of a single placement-set
  - E.g: *Do NOT place me on "this rack"*

- ***AND***ed placement-sets:
  - Returns intersection of two or more placement-sets
  - E.g.: *Place me on a machine "on this rack AND that which has this constraint"*

- ***OR***ed placement-sets:
  - Returns union of two or more placement-sets
  - E.g: Place me on
    - *"this host or that host"*
    - *"this rack or that rack"*

- ***Ordered OR***ed placement-sets:
  - This is similar to the OR operator but is a soft operator in that the scheduler will try to satisfy the first placement-set and if that fails, it proceeds to the next placement-set and so on, in the order specified

  - The syntax for Ordered OR expression is (placement-set-A with delay_to_next: X, placement-set-B).

  - If placement-set-B is not specified, we assume the default placement-set i.e. the wildcard/cluster placement.

Further attributes like maximum_allocations, delay etc can possibly apply on the *results* of the operator too! For e.g, you can apply conditions to form a placement-set A, combine that set with another set B using an operator O, and apply further conditions on the resultant set O(A, B).

The complex operators simply look at each of the placement-set involved one by one - note that it is not necessary for the scheduler internally to materialize the result of the operator (union / intersection / negation) completely to make the scheduling decision.

Using the set-operators described above and combining them in various orders, we can build arbitrarily complex placement strategies in the form of operator-trees. For e.g, one can say "*(rack=Rack1 OR rack=Rack2) AND (partition=GPU)"*

As we mentioned before while defining a placement-set, the *maximum-allocations* field can also be optionally specified on each placement-set. For example, imagine an app that wants to get a total of 100 allocations from {host1, host2, host3}, but does not want more than 50 of allocations from any of the individual hosts. This will translate into an operator-tree of the form "*(max_allocations = 100; ((host=host1, maximum_allocations=50) OR (host=host2, maximum_allocations=50) OR (host=host3, maximum_allocations=50)))".* Any successful allocation in a tree of placement-sets simply results in the updation of *maximum_allocation* counts all the way up into the hierarchy.

It is possible that the result of the application of these operators may be null-sets - for e.g. *"give me on this machine **and** on that machine"*. Depending on how complex the null-check validation is, the scheduler may choose to precompute the results of the application of the operator-tree and reject any of the requests that can *never* be satisfied.

Examples of Placement-Sets

The following is a non-exhaustive examples of various placement-sets that one can imagine.

1. **Site specific placement-sets:** At any site, a list of placement-sets usually are readily available for all user primarily as artifacts of cluster-setup and administrator configurations.
    - **Machine**: A machine is a simple placement-set with only one element - the host itself. This is either identified by the hostname or an IP-address.
    - ***Cluster*** (aka the *wild-card* placement). If a request specifies this as the placement-set, scheduler will place the container in an arbitrary location. This is identified either by the cluster-id or a wild-card (*) that indicates *this* cluster.
    - ***Admin configured placement-sets***: Today, admins already create a bunch of placement-sets based on physical (say hardware-level) properties, and whose membership can change over time. For e.g., an administrator can define a rack to be a set of machines that are colocated physically in a box - and machines can move in and out of a rack by admins/datacenter-ops depending on any site

changes. The following are a few examples of *admin configured placement-sets* that we consider for placement purposes.

- *Racks*: Typically defined by a rack-script, that administrators create to map every host to a specific *rack-name*.
- A *nodes partition* composing specific machines that satisfy a specific implicit condition
- A *racks partition:* A partition in the cluster composing of specific named racks. This notion is **not** yet implemented in YARN.

2. ***App specific placement-sets:*** Individual applications can make use of different site-specific placement-sets available and ***dynamically*** apply further explicit conditions and mix/match them to generate new placement-sets. They can similarly also obtain more ***derived*** placement-sets by applying conditions on other app specific placement-sets as follows:
    - A group of **machines** satisfying the condition “matching **specific node-attributes**”
    - A group of **machines** satisfying the condition “running **specific type of container of a specific app**”
    - A group of **machines** satisfying the condition “running **any container of a specific service application**”
    - A group of **racks** satisfying the condition “running **specific type of container of a specific app**”
    - A group of **partitions** satisfying the condition “running **specific app**”
3. Other types of placement-sets: So far, all the placement-sets we have seen are defined by an equality condition - host is x, partition is y, corresponding appid is z etc. One can imagine more dynamic conditions beyond equality - for e.g. any machine with > 100GB available memory. We don’t plan to design/implement such sets now, but present it as an example instance of a feature to think about, may be, far into the future.

(4.3.4) Time conditions

This part of the *ResourceRequest* identifies the “*when to schedule*” part of our story and answers questions about *when* allocations should should be returned, how long the allocations are supposed to be alive etc.

We have made a conscious choice to place *time-conditions* only at the top-level in the ResourceRequests. If users have complex requirements where they have to mix/match place, size etc with time within a single ResourceRequest, they should visit the section API design

choices for a discussion on non-use cases for a single ResourceRequest and potential workarounds.

Time Units

Going back to basics, the following is the dimensional space of *time-conditions* w.r.t various dimensions.

- *During a time interval*:
  - [T0, T1]: Between any time between T0 and T1
  - If T0 is not specified, we assume T0 to be current system-time
  - If T1 is not specified, we assume T1 to be a time far into the future (infinity)

- *At a specific time*:
  - We cannot realistically promise decisions at an exact time T0
  - So, we translate the (*At T0)* request to [T0, T0 + dT] where dT is the minimum confidence interval that scheduler can realistically promise users

Operators on time-dimensions

Similar to placement-sets, one can mix and match a bunch of ***disjoint*** *time-intervals* together by making use of some of the well-known set-operators.

- ***Null operator***:
  - A single time-interval
  - E.g:
    - Between *[2PM - 3PM]*
    - Between *[- 3PM]* i.e. anytime between *current-time and 3PM*
    - Between *[5PM -]* i.e. anytime *beyond 5PM*
- ***OR***ed time-intervals:
  - Returns union of two or more time-intervals
  - E.g: Between *[2PM - 3PM] OR [4PM-5PM or [9PM-10PM]]*

Other set-operators can simply be translated by the user into one of the above operators and so we do not support them in a first-class fashion

- ***Negation operator***
  - Negation of a single time-interval
  - E.g: *NOT between [2PM - 3PM].* This can simply be translated to *[* - 2PM] OR [3PM - *]*
- ***AND***ed time-intervals:
  - Returns intersection of two or more time-intervals
  - E.g: Between *[2PM - 5PM] AND [4PM-6PM]*. This is same as *[4PM - 5PM]*

Note that it is possible that the result of the application of these operators may be null-sets - for e.g. *"between [2PM - 2PM]"*. Scheduler will simply reject any of these requests that can *never* be satisfied. Clearly, past times cannot accepted and future times are essentially reservations.

Different time-dimensions

Now that we've defined the units of time specifications, we now proceed to different dimensions of time that app-developers may be interested in, categorized into two types:

- ***Resource allocation related time-dimensions***: Dimensions of this type control different time-intervals one can associate with the process of allocation itself.
    - *Allocation start-time:* This determines the time when an allocation should start getting satisfied by the scheduler. This is simply a trigger to start allocating resources, it doesn't guarantee that containers will definitely be allocated. If a guarantee is needed, users should couple this together with upfront reservations.
- ***Application Execution related time-dimensions***: This type of dimensions control different time-intervals one can associate with the actual work done after the allocation is successfully returned by the scheduler
    - *Execution duration-time*: Time taken for the entire containers' execution aka work-hours
    - *Execution deadline*: Determines the deadline from the application's perspective as to when the execution is supposed to finish. This is a hint to scheduler for planning.

In addition to these fundamental dimensions, there are several higher order constructs that are possible w.r.t allocations and their relationship with time:

- *Recurring time-schedules*: One can imagine requesting the platform to apply the allocation-time and execution-time dimensions repeatedly again and again.
- *Time-order dependencies*: One can have two *ResourceRequests* defined in such a way that they have time-order dependencies on each other - for e.g., allocate them one after another, allocation one after another with a gap of two hours etc.
- *Time-schedule failures*: What happens if the scheduler cannot give out allocations based on an agreed-upon time-schedule? Shall we simply mark the allocation-request or simply continue trying an allocation so as to salvage the deadline?

Time-based ResourceRequests and ApplicationMasters

Unlike placement-strategy, we don't quite define in this document how to think of designing the time conditions in the API nor do we hint at their underlying implementation.

Part of this is because of the need to consolidate it with existing work done at YARN-1051 YARN Admission Control/Planner: enhancing the resource allocation model with time which introduced some of the related notions but strictly operates in the context of client-side reservations-today.

Once we start imagining future reservations submitted **by** running applications with AMs, we will need to consider the problem of when to launch AM itself and make reservations.

For example, if we want to run a large mapreduce job with 10k mappers and 20k reducers at 11pm tomorrow, it will be a waste of resources if we launch the AM right now just for resource reservation later.

Instead, we can submit the regular ResourceRequests also as part of the *ApplicationSubmissionContext* for resource reservation, like so:

```
ApplicationSubmissionContext: {
  // Other fields...

  // ResourceRequests for AM container and task containers
  resource_requests_collection: [
    “application_master”: { // Resource request for AM

      priority: 0,
      allocation_size = 1G,
      maximum_allocations = 1,

      target_allocation_tags : {
        “component_name” : “AM”,
      },

      time_conditions: {
        allocation_start_time: [ 10:50 pm tomorrow - *]
      }
    },
    “12345”: { // Resource request for other task containers
      priority: 20,
      allocation_size = 10G,
      maximum_allocations = 10000,
```

```
      placement_strategies: {
         // .. Ignored
      },
      time_conditions: {
        allocation_start_time: [ 11:00 pm tomorrow - * ]
      }
    }
}
```

This way, the AppliationMaster itself will be scheduled to be launched at 10:50pm tomorrow, and other containers will be primed for launch from 11:00 pm.

(4.4) Allocation responses from scheduler to GUTS API requests

The Response to the applications largely remains unchanged, some points of note

- We will return ***Allocation-ID*** as part of the response
- From there, the application can figure out the corresponding priority, tag etc using either (a) its local scheduling state of the AM or (b) in case the AM doesn't want to manage local-state, it can invoke an API (to be added) in the scheduler to return the outstanding *ResourceRequests*
- Clearly, the scheduler may return multiple responses corresponding to the same ***Allocation-ID*** - except for the cases where the entire request is expecting a gang of allocations.

There is one very important unanswered question though. Today, as part of the response, the scheduler gives a hint called 'head-room'/ available-resources to the application so that the app can make certain decisions based on how much more resources can it get. The addition of placement-sets complicates the head-room notion a lot, please see the Conclusion section about a discussion this.

(5) Case studies

In this section, we give examples with varying levels of complexity so as to give a flavor of how the new API enables powerful requests and still keeps things simple enough.

An application with an affinity towards another application's components

Consider a stream-processing application that needs to write data to HBase, so it prefers to run on nodes where HBase RegionServers are already running - so as to exploit write-locality.

More concretely, let's look at an application with container-type / component name a "*stream_processor*" which *likes* to get placed near the containers running the storage service running on YARN with component-name "*hbase_region_server*" running under the YARN application *application_123456789_0005*.

This resource request tries to allocate on **hosts** which have allocations from application *application_123456789_0005* with component-name "*hbase_region_server*" first. After 2 mins, it relaxes to **racks** which have allocations from application *application_123456789_0005* with component-name "*hbase_region_server*". After 2 mins, it falls back to accepting any hosts in the cluster.

The corresponding scheduling request will look like the following:

```
resource_requests_collection:
  "567890": { // allocation_id

    priority: 123,

    allocation_size: 2G,
    maximum_allocations = 10,

    allocation_tags: {
      component_name: stream_processor,
    },

    placement_strategy: {
      ORDERED_OR [
        {
           // Default set-type, no need to specify
           placement_set_type: node,
           app_id: application_123456789_0005,
           target_allocation_tags : {
              component_name: hbase_region_server,
           },
           delay_to_next: 2 min or 100 scheduling_cycles
        },
        {
           // Delay to rack of following conditions
           placement_set_type: rack,
           app_id: application_123456789_0005,
           target_allocation_tags : {
              component_name: hbase_region_server,
```

```
          },
          delay_to_next: 2 min or 100 scheduling_cycles
        }
        {
          host: *
        }
      ]
    }
  }
```

A typical MapReduce Job

Below, we consider a typical MapReduce job needing 100 maps each of 2GB memory and 30 reducers each of 8GB. The input blocks are on host1, host2, host2.

```
resource_requests_collection: [
  “567890”: { // allocation_id

    priority: 123,

    allocation_size = 2G,
    maximum_allocations = 100,

    allocation_tags : {
      component_name : maps,
    }

    placement_strategy: {
      {
        OR: [
          {
            maximum_allocatons = 50,
            ORDERED_OR: [
              {
                OR: [
                  {
                    host: host1
                    maximum_allocations = 30
                  },
                  {
                    host: host2,
                    maximum_allocations = 30
                  },
```

```
                ],
                delay_to_next: 40 scheduling_cycles
              },
              {
                rack: /rack1,
                delay_to_next: 40 scheduling_cycles
              },
              {
                host: *,
              },
            ],
          },
          {
            maximum_allocatons = 50,
            ORDERED_OR: [
              {
                host: host3
                maximum_allocations = 50,
                delay_to_next: 40 scheduling_cycles
              },
              {
                rack: /rack2,
                delay_to_next: 40 scheduling_cycles
              },
              {
                host: *,
              },
            ],
          },
        ]
      },

  567891: {
    priority: 124,

    allocation_size = 8G,
    maximum_allocations = 30,

    allocation_tags : {
      “component_name” : “reducers”,
      “Workload_type”: “batch”
    }

    // Default placement-strategy is anywhere
  }
]
```

You should examine carefully how the example is laid out above. You will find that we do not specify any *maximum_allocations* against *rack1* at all - as the application really just cares about allocations on hosts, and the allocations on racks automatically follow. Any time an allocation is done on host1, the scheduler known that it needs one less on the rack too implicitly due to the object structure.

Anti-affinity case-studies

We now take up anti-affinity as defined under YARN-1042 as a special case-study to demonstrate the application of our new API proposal.

Anti-affinity is defined as the desire specified in a new *ResourceRequests* of an application to avoid places already specified by other *ResourceRequests*. We now look at several example use-cases in the context of anti-affinity.

Anti-affinity towards (nodes running containers of) another application

Assume you want to run an app that doesn't want to get allocated on hosts which are used by another specific application `application_123456789_0015`. Here's an example of how the *ResourceRequest* will look like:

```
resource_requests_collection:
  “567890”: { // allocation_id
    allocation_size: 2G,
    maximum_allocations = 10,

    placement_strategy: {
      NOT {
        // Default set-type is nodes
        // app_id is just like any other allocation_tag except that
        // it is a system recognized tag.
        app_id: application_123456789_0015
      }
    }
  }
```

As response to this request, scheduler will not give any resources to the requesting application on any hosts that are being used by *application_1234567890_0015*. For example, if one of the containers of *application_1234567890_0015* is on H1-H3, this application will not get allocations on H1-H3.

Self anti-affinity: Anti-affinity towards any hosts which have allocations from this app itself

If an app ***application_123456789_0007*** doesn't want to get resources allocated on the same host, it can make use of self anti-affinity. In the following request, the app tries to ask 10 allocations, expecting no two or more of these allocations on the same host.

```
resource_requests_collection:
  "567890": { // allocation_id
    allocation_size: 2G,
    maximum_allocations = 10,

    placement_strategy: {
      NOT {
        app_id: application_123456789_0007 // self reference
      }
    }
  }
```

Internally, the placement-sets will be updated by scheduler dynamically. Initially, there are no allocations to the application, so the placement-set is * (any hosts). Every time after an additional allocation is made to the application, scheduler internally blacklists the node just issued for the *ResourceRequest*. This is how the self anti-affinity is incrementally maintained so that the application doesn't get multiple allocations on the same host.

As you can see in this example, intra application affinity(or anti-affinity) is very similar to inter application affinity - all you need to do is to change the app_id.

Self anti-affinity: Anti-affinity towards any racks which have allocations from the app itself

Similar to the previous example, if an app ***application_123456789_0007*** doesn't want to get resources allocated on the same rack, it can make use of self anti-affinity. In the following request, the app tries to ask 10 allocations, and no two or more of these allocations on the same rack.

```
resource_requests_collection:
  "567890": { // allocation_id
    allocation_size: 2G,
    maximum_allocations = 10,

    placement_strategy: {
      NOT {
        placement_set_type: rack,
        app_id: application_123456789_0007 // self reference
```

```
      }
    }
  }
```

Changing *placement_set_type* to be a *rack* can create affinity/anti-affinity to racks.

Anti-affinity towards any hosts which have reducers running from the app itself

```
resource_requests_collection:
  “567890”: { // allocation_id
    allocation_size: 2G,
    maximum_allocations = 10,

    placement_strategy: {
      NOT {
        target_allocation_tags : {
          component_name: reducer,
        },
        app_id: application_123456789_0007 // self reference
      }
    }
  }
```

Similar to previous examples, adding *allocation_tags=reducer* to the request can result in the expected behavior.

An application with both affinity and anti-affinity requirements

One can imagine more examples in the same vein. For e.g., imagine a Zookeeper or a Storm application with the following requirements: *Given me 5 zookeeper containers, all on the same rack - I don't care which rack - all I care is that they're in the same rack - but only one per node.*

This is an example of an application which uses both affinity and anti-affinity requirements in the same strategy. We can simply use the AND operator to connect affinity-requirement (same rack) and anti-affinity (different hosts) as below:

```
resource_requests_collection:
  “567890”: { // allocation_id
    allocation_size: 2G,
    maximum_allocations = 10,

    placement_strategy: {
      AND [
```

```
          {
            placement_set_type: rack,
            app_id: application_123456789_0007 // self reference
          },
          NOT {
            placement_set_type: node
            app_id: application_123456789_0007 // self reference
          }
      ]
    }
  }
```

(6) API design choices and workarounds

New properties

In addition to the new categorization based on quantity / placement / time conditions, we introduced a couple of new fields as part of our design.

Allocation-ID is introduced primarily as a demuxer for different allocations - today applications cannot easily figure out for which original request are they getting the responses back. Note that this is not binding - applications can continue to completely ignore the returned allocation-id and use the allocation for any of their outstanding requests!

Allocation-Tags similarly are a convenience feature that simplify many type of scheduling specifications a lot. They can also be further used as bookkeeping hooks for requests even though the platform is likely going to put stringent limits on how many tags are applicable per request, per application etc.

Note that due to the top-level *allocation-tags* (and *application-id* as a special system-tag) being a part of a placement-strategy, many applications no longer need to know about the network layout at all in many situations! Clearly, applications like MapReduce still make use of the network topology, but a whole bunch of other services can be launched without doing so - this makes migration of such applications from cluster to cluster a little easier. For example, with the new API, a stream-processing application can simply declare its affinity towards the HBase application. Absent the new API, the application can still express its desire by first talking to the HBase application to figure out where HBase containers are running, get the corresponding host and rack locations and then specify them in turn via the old API to get locations that it wants for the sake of write-locality.

Design tradeoffs and workarounds

We have deliberately avoided many scheduling strategies that change dynamically as they seems too archaic at this moment that it may be reasonable to just ask the applications to orchestrate them outside of the scheduler, at least for now. In this section, we survey these ‘non use-cases’ for a single ResourceRequest and describe the workarounds.

We preclude the following use-cases throughout our API from being definable within a single ResourceRequest:

- Changing place
    - With time: *On host H1 at 2PM or host H2 at 10PM*
    - With size: *2GB container on Host1 or 5GB container on Rack2*
    - With max-concurrency: *on H1 with a width of 5 containers or H2 with a width of 10 containers*
    - With min-concurrency: *on R1 with a gang of 5 or R2 with a gang of 10*
- Changing time aka skylines
    - With max_allocations and min_concurrency/max_concurrency: *2PM-4PM 100 containers with a width of 20 and 4PM-6PM 50 containers with a width of 50*
- Changing time
    - With size: *On host H1: 2GB at 2PM or 4GB at 10PM*
    - With max_allocations: *4 allocations at 2PM or 10 allocations at 10PM*
- Changing size:
    - With max_concurrency: *2GB with a width of 100 containers or 4GB with a width of 50 containers*
    - With min_concurrency: *Gang of 4 * 2GB allocations or gang of 2 * 4GB allocations*
    - With max_allocations: *2GB of 10 allocations or 1GB of 20 allocations*
- Changing max_concurrency
    - With min_concurrency: *Gang of 4 allocations with a width of 100 containers or a Gang of 10 allocations with a width of 40 containers*
    - With max_allocations: *A maximum of 100 allocations with a width of 10 or a max of 50 with a width of 20*
- Changing min_concurrency and max_allocations: *A maximum of 100 allocations with a gang of 10 or a max of 50 with a gang of 20*

Some of the possibilities don’t have much real-life meaning and some of the above scenarios can be achieved by rewriting the scheduling request. We give a couple of examples below

- Changing place and size
    - *2GB container on Host1 or 5GB container on Rack2*
    - Instead submit two *ResourceRequests* one for *2GB on Host1* and another for *5GB on Rack2* at the same priority and orchestrate (reject) on the application side if both requests get satisfied at the same time
- Skyline / Changing time, max_allocations and min_concurrency/max_concurrency

- *2PM-4PM 100 containers with a width of 20 and 4PM-6PM 50 containers with a width of 50*
- Submit two separate Requests at different priorities. The second and further Requests that should come at 4PM etc, can simply have lower priority

To repeat, we have deliberately avoided encoding these scheduling strategies as first-class constructs in the API to keep the framework simple/reasonable enough for what we believe are more common use-cases.

API Unit Objects

As discussed before, Allocation-ID is the unit of the scheduling API: future requests mapped to an existing Allocation-ID will be treated by the scheduler as instructions to edit/modify/augment/cancel the original *ResourceRequest*.

Clearly, similar to the current API, update of only selected fields against a previously existing Allocation-ID will only update the object (as opposed to replacing it). For e.g, say a *ResourceRequest* first gets created with Allocation-ID "*76589*" and with placement_strategy "*{ host: *}*". A future *ResourceRequest* with the same Allocation-ID but with contents "*allocation_tags: { container_type: reducer}*" will only append the tags to the existing object. This is how one can replace parts of an object and is similar to how the existing per-record-deltas based protocol works.

Similarly, if one wishes to replace an entire *ResourceRequest* corresponding to a specific allocation-ID, they can simply cancel the corresponding *ResourceRequest* and submit a new one afresh.

API structures: Lossy vs lossless representation

As described before, one of the key design choices of the original scheduling API was to keep it as compact as possible given the use-cases at the time (MapReduce). In today's YARN API, there is a information-loss that we traded off for the sake of compact representation over the wire and in memory. Most of the upcoming use-cases are stretching the limits of this lossy representation.

Digging more, in the previous world, you may have known that the scheduler implicitly assumes the relationships between container-counts corresponding to node and rack and automatically updates (decrements) them when an allocation happens. This completely makes some types of requests impossible to specify - for example *"OR { host: H1, maximum_allocations: 20}, { rack: R1, maximum_allocations:40}"* where the host and rack requirements are completely independent - an artifact of lossiness.

The GUTS API now lets users to avoid the lossiness completely!

- If the user deliberately needs a relationship between node and rack counts, he/she can wrap them in an object and set the corresponding *maximum_allocations.*
    - In our MapReduce example in the case-studies section, you will find that we do not specify any *maximum_allocations* against *rack1* at all - as the application really just cares about allocations on hosts, and the allocations on racks automatically follow. Any time an allocation is done on host1, the scheduler known that it needs one less on the rack too implicitly due to the object structure.
- If the user deliberately **does not** need a relationship between node and rack counts, he/she can keep them as independent objects and set the corresponding *maximum_allocations.*
    - With this, examples like *"OR { host: H1, maximum_allocations: 20}, { rack: R1, maximum_allocations:40}"* become possible.

This also keeps the scheduler side of the story simple too. As we mentioned in previous section, overall, any successful allocation in a tree of placement-sets simply results in the updation of *maximum_allocation* counts all the way up into the hierarchy.

API structures: resource vs task-centric models

Further, on first-look, it may look like our new proposal is a hybrid model between the either extremes of resource vs task-centric APIs that the YARN community has discussed a lot in the past. However, looking at a deeper level, it will be clear that we still restrict the GUTS API to be a resource-centric (albeit much more powerful) API even though per-task requests are *not* entirely obviated anymore. We are well aware of the original design discussions around avoiding task-centric APIs and hence strive to limit exposing the extremes of task-centricity by explicitly putting several limits on the *ResourceRequest*. For e.g., we will definitely limit the total number of outstanding *ResourceRequests* per application (or unique outstanding Allocation-IDs) to be on the order of the cluster's size (node-count). The intuition here is that even if the application needs very convoluted and specific placement strategies, the most expensive way to define them is to specify them one per node in the cluster.

Dynamic nature of the placement-groups

In most of our examples, we have demonstrated how requests that depend on other placement-sets can be implemented at a high level in the scheduler - mainly by way of set operations. Internally, the scheduler computes a set of nodes that can be used by each *ResourceRequest*. When the dependent placement-sets don't change during the life-cycle of

this allocation (say predefined racks/partitions etc), it is a simple job for the scheduler. It becomes challenging for the scheduler when the set dynamically changes though.

For instance, in the example for section anti-affinity towards (nodes running containers of) another application, we looked at a request (say of app 2000) that has anti-affinity towards *application_1234567890_0015.* The initial scheduling of resources is easy - the scheduler simply avoids machines that are running containers of *application_1234567890_0015.* What should the semantics for app 2000 be when the placement-set corresponding to *application_1234567890_0015* changes due to expansion or failures? There are two approaches here:

- Scheduler ignores any further changes after the initial decision - this is a simple decision but pushes semantics to applications and complicates users' life.

- Scheduler tracks all changes after the initial decision: The placement-set can be updated if any resource allocation / release happens.

    - For instance, if *application_1234567890_0015* releases all containers on a host, scheduler can mark the host to be usable by app 2000 again.

    - Similarly, if *application_1234567890_0015* gets a new allocation on a different host, e.g. H5, this application can no longer get any new resources allocated on H5.
        - What happens to existing containers that are running already on H5 ?!

    - In either case, keeping the placement-sets up to date all the time can get really expensive in a cluster with lots of applications.

- There are further possibilities about the direction and dynamics of affinity/anti-affinity

    - One can imagine unidirectional or bidirectional affinity/anti-affinity. With more than two entities involved, this can become multi-directional anti-affinity like in the case of a multi-way mutex.

    - If there is a mixture of affinity, anti-affinity requirements, the scheduler's job will become harder. For instance, *app A* may have an affinity towards *app B* but anti-affinity towards *app C,* at the same time *B* and *C* may have affinity towards each other.

- Further, a single allocation can specify anti-affinity towards components of N other applications. The more applications have to be accounted for in the placement-set calculation, the longer it is going to take scheduler to make decisions. It is likely that we

will put an implementation of limit of being able to specify affinity/anti-affinity towards allocations that belong to only a few (2-3) other applications.

(7) General notes, Open issues & Conclusion

Applying the same concepts at the app-level

Note that many concepts that we have introduced per-ResourceRequest make sense even at the application-level, here we layout a few of them without really establishing whether we can go after them during our implementation:

- *Application-Tags* are analogous to Allocation-Tags per allocation, but not directly relevant in the context of scheduling
- Quantity Conditions
    - *Maximum-concurrency*: We can imagine a maximum-concurrency applicable across all running containers for the app. MAPREDUCE-5583 enabled the same feature but only in the MapReduce land.
    - *Maximum number of allocations:* This can act as an upper limit on the total number of allocations returned to an application during its entire life-time.
- Placement Strategy
    - This can be an application-level strategy that provides either a default strategy or limits the universal placement-set (*scope)* of all the app's *ResourceRequests* for e.g. make sure no allocation within this application can go outside of *this* partition.
- Time conditions
    - *Allocation start-time* per allocation is analogous to reservation (YARN-1051) trigger time.
    - *Execution duration-time*: Time taken for the entire containers' execution aka work-hours for all of the application - also called as life-time.
    - *Recurring time-schedules*: One can imagine requesting the platform to apply the allocation-time and execution-time dimensions repeatedly again and again - aka a recurring workflow of the application.
    - *Time-order dependencies*: One can have two applications defined in such a way that they have time-order dependencies on each other - for e.g., start running them one after another, or one after another with a gap of two hours etc.

Head Room response to applications

As you can expect, there a few key areas that are still open for more thought on how we solve generally. - for e.g. how *application head-room* transforms in this generalized request model. As mentioned before, today, the scheduler gives a hint as part of the response called 'head-room'/ available-resources to the application so that the app can make certain decisions based on how much more resources can it get. The addition of placement-sets complicates the head-room

notion a lot. The dynamic nature of the placement-sets, depending on the state of N other applications just makes it worse.

This is largely still an open problem to solve.

Coexistence with current API

There are multiple possibilities in actually implementing the GUTS API. The inertial approach is to simply extend existing *ResourceRequest*. In fact, implementing some of the newer concepts like *Allocation-ID, allocation-tags etc* are easy to implement by just augmenting the current API.

However, augmenting the existing *ResourceRequest* to support our placement-sets, operators etc is not possible. As summarized in the existing-characteristics section (2), today's *ResourceRequest* has many hard traits - it is keyed by requested resource-size and priority, only two types of resource-names (racks, nodes) are possible and there is inherent linkage between the node/rack counts and the wild-card (*) num-containers. These assumptions are not easily broken without also breaking compatibility of the existing scheduling behavior.

Compatibility

We need some sort of backwards compatibility story given the new proposal. Clearly, we cannot modify each and every YARN application to support the new GUTS API, even though it may be desirable to do so for some of the new applications. There are four actors here: Server-side implementation, server-side API, client-libraries implementation, client-side API aka application changes. We have multiple options in this regard

- Keep it only on the server-side implementation in the beginning
- Make the server-side implementation, add a new server-side API, expose it as an alternative API to clients
    - Modify the implementation of existing client libraries to funnel through the new server side API.
    - Add new APIs to the client libraries (which speak the new native language with the server side) and make them usable by new/existing applications that wish to leverage the power of the new model directly.

As a first step, even before we expose this in the API, we lean towards making the scheduler internally use and implement the GUTS API instead of the existing *ResourceRequest.* This is because the new API is richer and a more general form of existing ResourceRequest. In many cases like with many of the non-MapReduce use-cases, it can actually result it more compact internal representations.

Scheduling Cost

Without even going into the exact implementation details, one can clearly see that our proposal (with new concepts like arbitrarily complex placement-trees, server-side support for features like gang-scheduling) will introduce a gamut of scheduling paths that are much more expensive compared to existing code-paths for simple host/rack matching.

It is likely that we will have to move some or most of these to be decisions made offline instead of the inline heartbeat-triggered scheduling-cycles today.

Dependencies

There may be a need for being to able to specify further dependencies that are out of scope of this document. Additional type of dependency/ordering may exist between two or more ResourceRequests (beyond priority-order).

Alternative approaches: Distance measures

Even with our GUTS API, we continue to model different entities in a cluster without explicitly modeling their distances. For e.g, we continue to assume that placement on the same node is implicitly *better* than on the same rack, if affinity is desired, without explicitly modeling the distance between nodes / racks etc.

The reason we bring this up is because explicit modeling of distances can lead to a different way of API specifications. For instance, instead of specifying affinity towards a node or a rack, an application can ask that it be placed at a distance not more than X units away from its data or other dependent application. Similarly, requests all allocations on the same node can be specified by a distance of zero units. Taking this further, higher order distance functions like ***closer (packing)*** *vs* ***farther (spreading)*** can be introduced instead of boolean operators of affinity/anti-affinity towards placement-sets (with predefined and implicit distance measures).

This is left as a future exercise, potentially augmenting our updated proposal using the GUTS API.

Higher order API utilities / syntactic sugar

Our APIs under placement-strategy and time-conditions are modeled using boolean expressions and as such are at the same time very powerful and a bit low-level. If specific features like affinity and anti-affinity are more common than others, we can add syntactic sugar to make them as first-class concept in the API.

Similar arguments can be made about the time-intervals - we can support *before* / *after* like phrases instead of limiting ourselves to time-interval specs.

In the java API itself, we can imagine a more fluent APIs (vs today's implied object mangling) for easier use. For e.g., one can say something like

```
new PlacementStrategy()
    .avoid(new Place("app1").on("host1"))
    .with(new Place("RegionServer").on(host1))
    .waitFor("2min")
    .on("rack");
```

Conclusion

This concludes our attempt at a unified theory of scheduling strategies for now, with a clear understanding that there is a lot more left for discussion and further refining to come.

We created this document mainly to coalesce thoughts similar to ours that we heard during interactions in the community over a long period of time. We hope to trigger a conversation towards a design with more comprehensive details as needed and lesser gaps in it.