Gang Scheduling Implementation

Change log

Change Date	*Editors*	*Description*
2020-11-25	Wilfred Spiegelenburg	Initial document
2020-11-26	Wilfred Spiegelenburg	Allocation release definition, order change
2020-11-27	Wilfred Spiegelenburg	Sequence diagram
2020-12-12	Wilfred Spiegelenburg	Application submit and cleanup

2020-12-15	Wilfred Spiegelenburg	AllocationRelease message and cleanup sequence
2020-12-24	Wilfred Spiegelenburg	Allocation changes for recovery
2021-02-15	Wilfred Spiegelenburg	Placeholder allocation and ask cleanup

YUNIKORN-2 describes a new format for scheduling applications by taking into account the overall demand the application will have. It guarantees the specified resources for the application by reserving the resources.

There are two parts to this implementation:

- Kubernetes Shim
- Core and scheduling

This document describes the implementation on the core side.

Goals

Define the following points:

1. Define changes required for the shim to core communication (scheduler interface)
2. Scheduler storage object changes
3. Scheduler logic changes

Non Goals

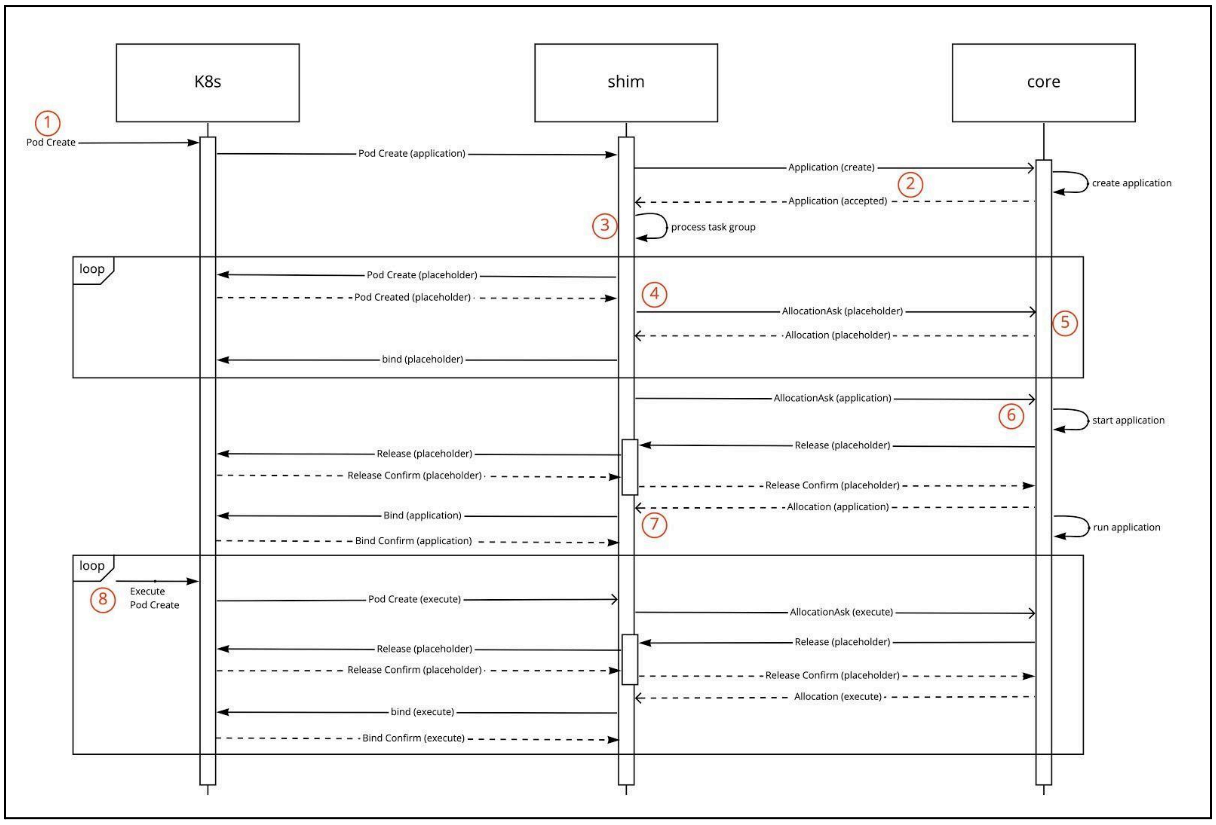
Excluding the following major points:

1. Kubernetes shim side implementation
2. Generalised preemption on the core side

Generic flow

As described in the YUNIKORN-2 design documentation we have the following flow. The flow is triggered by a pod that is submitted which triggers the application creation. This first pod is in the case of a Spark application, the driver pod. In case the flow is triggered from the creation of an application CRD there will not be a first pod. This is however outside of the core scheduling logic. From the core side there should be no difference between the two cases. More details are in the chapter on the Scheduler logic changes.

The flow of an application submitted. The numbers in the diagram correspond to the description below the diagram.



Combined flow for the shim and core during startup of an application:

- An application is submitted with TaskGroup(s) defined. (1)
- The shim creates the application and passes the application to the core. (2)
- The shim creates placeholder pods for each of the members of the TaskGroup(s) (3)
- The pods are processed and passed to the core, as per the normal behaviour, as AllocationAsks for the application with the correct info set. (4)
- The placeholder AllocationAsk's are scheduled by the core as if they were normal AllocationAsk's. (5)
- All Allocations, even if they are the result of the placeholder AllocationAsks being allocated by the scheduler, are communicated back to the shim.
- The original real pod is passed to the core as an AllocationAsk. (6)
- After the real pod and all all the placeholder pods are scheduled the shim starts the real pod that triggered the application creation. (7)

After the first, real, pod is started the following pods should all be handled in the same way (8):

- A real pod is created on k8s.
- The pod is processed and an AllocationAsk is created.
- The scheduler processes the AllocationAsk (more detail below) and replaces a placeholder with the real allocation.

Application submit handling

Total placeholder size

The application if requesting one or more TaskGroups should provide the total size of all the TaskGroup members it is going to request. The total resource size is required for the case that the application is scheduled in a queue with a resource limit set.
The value is important for three cases:

1. gang request is larger than the queue quota
2. start of scheduling reservations
3. resource pressure while scheduling reservations

Further detail will be given below in scheduling in queues with a quota set

The information passed on from the shim should be part of the AddApplicationRequest. Detailed information on the build up of the taskGroup(s) or the number of members are not relevant. The total resource requested by all taskGroup members is calculated using:

$$taskgroup\ Ask\ =\ \#\ members\ \ *\ \ resource\ per\ member$$

$$total\ placeholderAsk\ = \sum\ (taskgroups\ Ask)$$

This total placeholderAsk is added as an optional field to the AddApplicationRequest message. The calculation can be made by the shim based on the CRD or annotation provided in the pod description.

If the placeholderAsk is larger than the queue quota set on the queue the application must be rejected. This rejection is based on the fact that we cannot in any way honor the request For all other cases the application is accepted and will be scheduled as per normal.

Handling queue with a FAIR sort policy

If an application is submitted to a queue that has a FAIR sort policy set it must be rejected. Queue sorting for the queue that an application with gang requests runs in must be set to FIFO or StateAware.

Other queue policies cannot guarantee that there is only one New application processed at a time. In the case of the FAIR policy we could be allocating multiple New applications at the same time making quota management impossible to enforce. The other side effect of using FAIR as a policy could be that we get multiple applications with only a partial allocated guarantee.
Auto scaling can be triggered due to the fact that the core can not place the placeholders on any node. In case the queue would use the FAIR sorting this could lead to other applications taking the scaled up nodes instead of the placeholders again breaking the gang.

Scheduling in queues with a quota set

The main case already described above is handling a total placeholder request size that is larger than the quota set on the queue. When the application is submitted we can already assess that we cannot satisfy that requirement and reject the request.

In the case that the total placeholder ask does fit in the queue we should not start scheduling until there are enough resources available in the queue to satisfy the total request. However this does not stop scheduling of other applications in the queue(s). Applications that are already running in the queue could ask for more resources. From an application perspective there is no limit set on the resource it can request. The gang defined on the application is a guaranteed number of resources, not a maximum number of resources the application can request.

This is complicated by the fact that we have a queue hierarchy. There is the possibility that the quota is not set directly on the queue the application is running. It could be set on one of the parent queues. This case could become complex and we need to make sure that we keep in mind that we could live lock the scheduling.

In this first phase we should focus on the case that the gang resources requested are also the maximum number of resources the application will request. When we look at the queues we should focus on a single queue level with quotas.
These two assumptions are correct for the spark use case without dynamic allocation using a dynamic mapping from a namespace to a queue.
Furthermore we assume that the quota set on the queue can be totally allocated. If the cluster does not have enough resources the cluster will scale up to the size needed to provide all queues with their full quota.

The follow up should add further enhancements for deeper hierarchies and dynamic allocation support. This could also leverage preemption in certain use cases, like preempting allocations from applications over their total gang size.
Further enhancements could be added by allowing specifying the time and application will wait for the placeholders to be allocated or the time to start using the held resources.

Scheduler logic changes

The scheduler logic change needs to account for two parts of cycle:

- The placeholder asks and their allocation.
- The allocation replacing the placeholder.

The basic assumption is that all pods will generate a placeholder pod request to the core. This includes the pod that triggered the application creation if we do not use the application CRD. This assumption is needed to make sure that the scheduler core can behave in the same way for both ways of submitting the application. The placeholder pods must be communicated to the core before the real pod.

Queue sorting for the queue that the application runs in must be set to *FIFO* or *StateAware*. Other queue policies cannot guarantee that there is only one *New* application processed at a

time. In the case of the *FAIR* policy we could be allocating multiple *New* applications at the same time making quota management impossible to enforce. The other side effect of using *FAIR* as a policy could be that we get multiple applications with only a partial allocated guarantee.
Auto scaling can be triggered due to the fact that the core can not place the placeholders on any node. In case the queue would use the *FAIR* sorting this could lead to other applications taking the scaled up nodes instead of the placeholders again breaking the gang.

Changes for the placeholder AllocationAsks are the first step. As part of the creation of the application the AllocationAsks get added. The addition of an AllocationsAsk normally will trigger the application state change as per the scheduling cycle. It moves the Application from a *New* state to an *Accepted* state. This is as per the current setup, and does not change.

However in the case that the AllocationAsk has the *placeholder* flag set the allocation should not trigger a state change, the application stays in *Accepted* state. AllocationAsks are processed until the application has no pending resources. AllocationAsks that do not have a *placeholder* flag set should be ignored as a safety precaution. All resulting Allocations for the placeholder pods are confirmed to the shim as per the normal steps. This process continues until there are no more placeholder pods to be allocated.

The shim at that point should create the AllocationAsk for the real pod(s) that it has buffered. The core cannot and must not assume that there is only one task group per application. The core is also not in the position to assume that it has received all AllocationAsks that belong to the task group if option 1 as described above is used by a shim. This is also why we have the assumption that every pod creates a placeholder request to the core.

The second change is the replacement of the placeholder pods with the real pods. The shim creates an AllocationAsk with the *taskGroupName* set but the *placeholder* flag is not set.

The process described here lines up with the process for generic pre-emption. An allocation is released by the core and then confirmed by the shim. For gang scheduling we have a simple one new to one release relation in the case of pre-emption we can use the same flow with a one new to multiple release relation.

The scheduler processes the AllocationAsk as follows:

1. Check if the application has an unreleased allocation for a placeholder allocation with the same *taskGroupName.* If no placeholder allocations are found a normal allocation cycle will be used to allocate the request.
2. A placeholder allocation is selected and marked for release. A request to release the placeholder allocation is communicated to the shim. This must be an async process as the shim release process is dependent on the underlying K8s response which might not be instantaneous.
   NOTE: no allocations are released in the core at this point in time.
3. The core “parks” the processing of the real AllocationAsk until the shim has responded with a confirmation that the placeholder allocation has been released.
   NOTE: locks are released to allow scheduling to continue

4. After the confirmation of the release is received from the shim the “parked” AllocationAsk processing is finalised.
5. The AllocationAsk is allocated on the same node as the placeholder used.
    - On success: a new Allocation is created.
    - On Failure: try to allocate on a different node, if that fails the AllocationAsk becomes unschedulable triggering scale up.

   The removal of the placeholder allocation is finalised in either case. This all needs to happen as one update to the application, queue and node.
6. Communicate the allocation back to the shim (if applicable, based on step 5)

Application completion

Application completion has been a long standing issue. Currently applications do not transition to a *completed* state when done. The current states for the application are documented here. However at this point in time an application will not reach the *completed* state and will be stuck in *waiting*.
This provides a number of issues specifically around memory usage and cleanup of queues in long running deployments.

Definition

Since we cannot rely on the application, running as pods on Kubernetes, to show that it has finished we need to define when we consider an application *completed*. At this point we are defining that an application is *completed* when it has been in the *waiting* state for a defined time period. An application enters the waiting state at the time that there are no active allocations (allocated resources > 0) and pending allocation asks (pending resources > 0).

The transition to a *waiting* state is already implemented. The time out of the *waiting* state is new functionality.

Placeholders are not considered active allocations. Placeholder asks are considered pending resource asks. These cases will be handled in the Clean up below.

Clean up

When we look at gang scheduling there is a further issue around unused placeholders, placeholder asks and their cleanup. Placeholders could be converted into real allocations at any time there are pending allocation asks or active allocations.
Placeholder asks will all be converted into placeholder allocations before the real allocations are processed.

Entry into the *waiting* state is already handled. If new allocation asks are added to the application it will transition back to a *running* state. At the time we entered the waiting state. there were no pending requests or allocated resources. There could be allocated placeholders.

For the entry into the *waiting* state the application must be clean. However we can not guarantee that all placeholders will be used by the application during the time the application runs. Transitioning out of the *waiting* state into the *completed* state requires no (placeholder) allocations or asks at all. The second case that impact transitions is that not all placeholder asks are allocated and the application thus never requests any real allocations. These two cases could prevent an application from transitioning out of the *accepted* or the *waiting* state.

Processing in the core thus needs to consider two cases that will impact the transition out of specific states:

1. Placeholder asks pending (exit from *accepted*)
2. Placeholders allocated (exit from *waiting*)

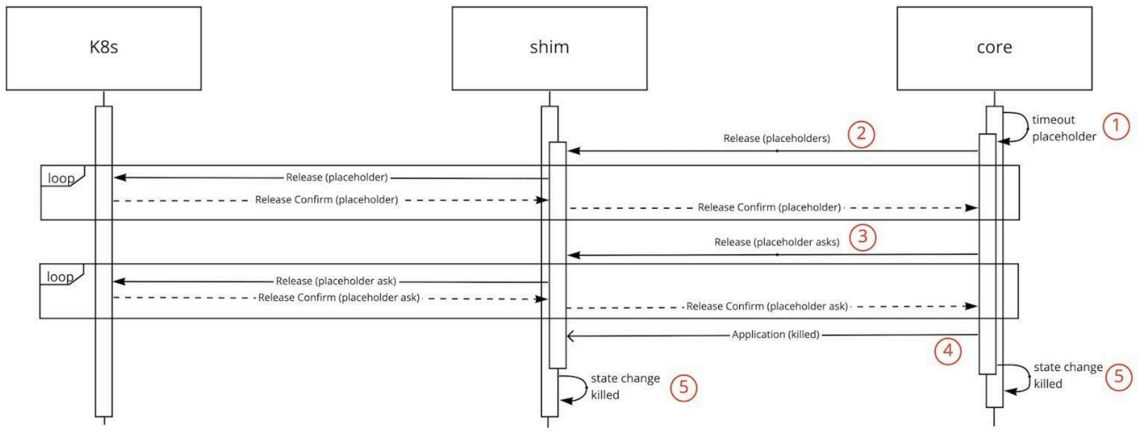
Placeholder asks pending:
Pending placeholder asks are handled via a timeout. An application must only spend a limited time waiting for all placeholders to be allocated. This timeout is needed because an application's partial placeholders allocation may occupy cluster resources without really using them.
An application could be queued for an unknown time, waiting for placeholder allocation to start. The timeout for placeholder asks can thus not be linked to the creation of the application or the asks. The timeout must start at the time the first placeholder ask is allocated.
The application cannot request real allocations until all placeholder asks are allocated. A placeholder ask is also tracked by the shim as it represents a pod. Releasing an ask in the core requires a message to flow between the core and shim to release that ask. However in this case the timeout for allocating placeholder asks triggers an application failure. When the timeout is triggered and placeholder asks are pending the application will transition from the state it is in, which can only be *accepted*, to *killed*.

The application state for this case can be summarised as:
- Application status is *accepted*
- Placeholder allocated resource is larger than zero, and less than the *placeholderAsk* from the *AddApplicationRequest*
- Pending resource asks is larger than zero

Entering into the *killed* state must move the application out of the queue automatically.
The state change and placeholder allocation releases can be handled in a single UpdateResponse message. The message will have the following content:

- *UpdatedApplication* for the state change of the application
- one or more *AllocationRelease* messages, one for each placeholder, with the *TerminationType* set to TIMEOUT
- one or more AllocationAskRelease messages with the *TerminationType* set to TIMEOUT

The shim processes the AllocationAskRelease messages first, followed by the *AllocationResponse* messages, and finally the *UpdatedApplication* message. The application state change to the *killed* state on the core side is only dependent on the removal of all placeholders pods, not on a response to the *UpdatedApplication* message.



K8s
shim
core
timeout placeholder
1
Release (placeholders)
2
loop
Release (placeholder)
Release Confirm (placeholder)
Release Confirm (placeholder)
Release (placeholder asks)
3
loop
Release (placeholder ask)
Release Confirm (placeholder ask)
Release Confirm (placeholder ask)
Application (killed)
4
state change killed
5
state change killed
5

Combined flow for the shim and core during timeout of placeholder:

- The core times out the placeholder allocation. (1)
- The placeholder Allocations removal is passed to the shim. (2)
- All placeholder Allocations are released by the shim, and communicated back to the core.
- The placeholder AllocationAsks removal is passed to the shim. (3)
- All placeholder AllocationAsks are released by the shim, and communicated back to the core.
- After the placeholder Allocations and Asks are released the core moves the application to the killed state removing it from the queue (4).
- The state change is finalised in the core and shim. (5)

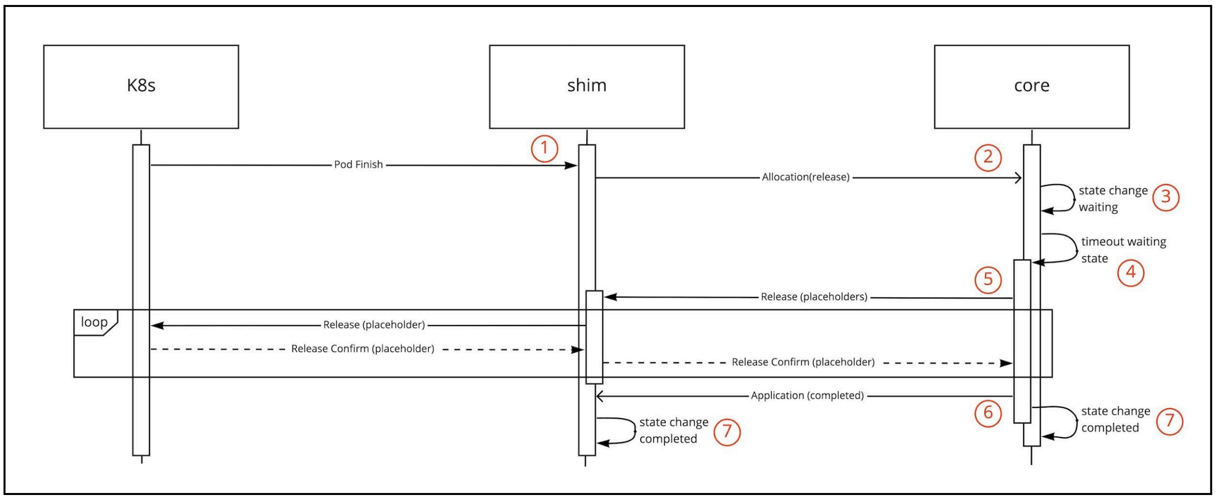
Allocated placeholders:
Leftover placeholders need to be released by the core. The shim needs to be informed to remove them. This must be triggered on entry of the *completed* state. After the placeholder release is requested by the core the state transition of the application can proceed. The core will process the *AllocationRelease* messages for placeholder allocations that come back from the shim with the *TerminationType* set to TIMEOUT as normal without triggering a state change.

The state change and placeholder allocation releases can be handled in a single UpdateResponse message. The message will have the following content:

- *UpdatedApplication* for the state change of the application
- zero or more *AllocationRelease* messages, one for each placeholder, with the *TerminationType* set to TIMEOUT

The shim processes the *AllocationResponse* messages first followed by the *UpdatedApplication* message. The application state change to the *completed* state on the core side is only dependent on the removal of all placeholders pods, not on a response to the *UpdatedApplication* message.

Entering into the *completed* state will move the application out of the queue automatically. This should also handle the case we discussed earlier around a possible delayed processing of requests from the shim as we can move back from *waiting* to *running* if needed. A *completed* application should also not prevent the case that was discussed around cron like submissions using the same application ID for each invocation. A *completed* application with the same application ID must not prevent the submission of a new application with the same ID.



Combined flow for the shim and core during cleanup of an application:

- A pod is released at the Kubernetes layer. (1)
- The shim passes the release of the allocation on to the core. (2)
- The core transitions the application to a waiting state if no pending or active allocations. (3)
- The waiting state times out and triggers the clean up. (4)
- The placeholder Allocations removal is passed to the shim. (5)
- All placeholder Allocations are released by the shim, and communicated back to the core.
- After all placeholders are released the core moves the application to the completed state removing it from the queue (6).
- The state change is finalised in the core and shim. (7)

Application recovery

During application recovery the placeholder pods are recovered as any other pod on a node. These pods are communicated to the core by the shim as part of the node as an existing allocation. Existing allocations do not have a corresponding *AllocationAsk* in the core. The core generates an *AllocationAsk* based on the recovered information.
For gang scheduling the *AllocationAsk* contains the *taskGroupName* and *placeholder* flag. During recovery that same information must be part of the *Allocation* message. This is due to the fact that the same message is used in two directions, from the RM to the scheduler and vice versa means we need to update the message and its processing.

If the information is missing from the *Allocation* message the recovered allocation will not be correctly tagged in the core. The recovered allocation will be seen as a regular allocation.

This means it is skipped as part of the normal allocation cycle that replaces the placeholders.
The logic change only requires that the recovery of existing allocations copies the fields from the interface message into the allocation object in the core.

Interface changes

Multiple changes are needed to the communication between the shim and the core to support the gang information needed.
An application must provide the total size of the placeholder requests to prevent accepting an application that can never run.
The current object that is sent from the shim to the core for allocation requests is defined in the AllocationAsk. The Allocation, as the result message passed back from the scheduler core does not change. For recovery, which uses the same Allocation message, from the shim to the core, however must contain the gang related fields. Gang related fields must be added to both messages.
The allocation release request and response request need to support bidirectional traffic and will need to undergo major changes.

AddApplication

The AddApplicationRequest message requires a new field to communicate the total placeholder resource request that will be requested. The field is used to reject the application if it is impossible to satisfy the request. It can also be used to stop the core from scheduling any real pods for that application until all placeholder pods are processed.

In patched message form that would look like:

```
message AddApplicationRequest {
...
  // The total amount of resources gang placeholders will request
  Resource placeholderAsk = 7;
...
}
```

AllocationAsk

The first part of the change is the base information for the task group. This will require an additional optional attribute to be added. The content of this optional attribute is a name, a string, which will be mapped to the name of the task group. The field can be present on a real allocation and on a placeholder.

Proposed name for the new field is: *taskGroupName*

To distinguish normal AllocationAsks and placeholder AllocationAsks a flag must be added. The flag will never have more than two values and thus maps to a boolean. As the default value for a boolean is *false* the field should show the fact that it is an AllocationAsk that represents a placeholder as true.

Proposed name for the field is: *placeholder*

In patched message form that would look like:

```
message AllocationAsk {
...
  // The name of the TaskGroup this ask belongs to
  string taskGroupName = 10;
  // Is this a placeholder ask (true) or a real ask (false), defaults to false
  // ignored if the taskGroupName is not set
  bool placeholder = 11;
...
}
```

The last part of the task group information that needs to be communicated is the size of the task group. This does not require a change in the interface as the current AllocationAsk object can support both possible options.
Requests can be handled in two ways:

A. Each member of the task group is passed to the core as a separate AllocationAsk with a maxAllocations, or the ask repeat, of 1
B. The task group is considered one AllocationAsk with a maxAllocations set to the same value as minMember of the task group information.

With option A the shim will need to generate multiple AllocationAsk objects and pass each to the core for scheduling, Each AllocationAsk is linked to one pod. Option B will only generate one AllocationAsk for all placeholder pods. Option B requires less code and has less overhead on the core side. However the logic on the shim side might be more complex as the returned allocation needs to be linked to just one pod.

Proposal is to use option: A

Allocation

Similar to the change for the *AllocationAsk* the *Allocation* requires additional optional attributes to be added. The new fields distinguish a normal Allocation and placeholder Allocations on recovery. The same rules apply to these fields as the ones added to the *AllocationAsk*.
The content of this optional attribute is a name, a string, which will be mapped to the name of the task group. The field can be present on a real allocation and on a placeholder.

Proposed name for the new field is: *taskGroupName*

The flag will never have more than two values and thus maps to a boolean. As the default value for a boolean is *false* the field should show the fact that it is an Allocation that represents a placeholder as true.

Proposed name for the field is: *placeholder*

In patched message form that would look like:

```
message Allocation {
...
  // The name of the TaskGroup this allocation belongs to
  string taskGroupName = 11;
  // Is this a placeholder allocation (true) or a real allocation (false), defaults to false
  // ignored if the taskGroupName is not set
  bool placeholder = 12;
...
}
```

AllocationRelease Response and Request

The name for the messages are based on the fact that the release is always triggered by the shim. In case of preemption and or gang scheduling the release is not triggered from the shim but from the core. That means the message name does not cover the usage. A response message might not have an associated request message. It could be used to indicate direction but that is in this case confusing.

When a release is triggered from the core, for preemption or the placeholder allocation, a response is expected from the shim to confirm that the release has been processed. This response must be distinguished from a request to release the allocation initiated by the shim. A release initiated by the shim must be followed by a confirmation from the core to the shim that the message is processed. For releases initiated by the core no such confirmation message can or must be sent. In the current request message there is no way to indicate that it is a confirmation message.

To fix the possible confusing naming the proposal is to merge the two messages into one message: *AllocationRelease*.
The *AllocationReleaseRequest* is indirectly part of the *UpdateRequest* message as it is contained in the *AllocationReleasesRequest*. The *AllocationReleaseResponse* is part of the *UpdateResponse* message. The flow-on effect of the rename and merge of the two messages is a change in the two messages that contain them. The message changes for *UpdateResponse* and *AllocationReleasesRequest* are limited to type changes of the existing fields.

Message	Field ID	Old type	New type
UpdateResponse	3	AllocationReleaseResponse	AllocationRelease
AllocationReleasesRequest	1	AllocationReleaseRequest	AllocationRelease

In patched message form that would look like:

```
message UpdateResponse {
...
  // Released allocation(s), allocations can be released by either the RM or scheduler.
  // The TerminationType defines which side needs to act and process the message.
  repeated AllocationRelease releasedAllocations = 3;
...
}
```

```
message AllocationReleasesRequest {
  // Released allocation(s), allocations can be released by either the RM or scheduler.
  // The TerminationType defines which side needs to act and process the message.
  repeated AllocationRelease releasedAllocations = 1;
...
}
```

The merged message *AllocationRelease* will consist of:

Field name	Content type	Required
partitionName	string	yes
applicationID	string	no
UUID	string	no
terminationType	*TerminationType*	yes
message	string	no

Confirmation behaviour of the action should be triggered on the type of termination received. The core will confirm the release to the shim of all types that originate in the shim and vice versa.
A confirmation or response uses the same *TerminationType* as was set in the original message. An example of this is a pod that is removed from K8s will trigger an *AllocationRelease* message to be sent from the shim to the core with the TerminationType STOPPED_BY_RM. The core processes the request removing the allocation from the internal structures, and when all processing is done it responds to the shim with a message using the same *TerminationType*. The shim can ignore that or make follow up changes if needed.
A similar process happens for a release that originates in the core. Example of the core sending an *AllocationRelease* message to the shim using the *TerminationType* PREEMPTED_BY_SCHEDULER. The shim handles that by releasing the pod identified and responds to the core that it has released the pod. On receiving the confirmation that the pod has been released the core can progress with the allocation and preemption processing.

In patched message form that would look like:

```
message AllocationRelease {
  enum TerminationType {
    STOPPED_BY_RM = 0;
    TIMEOUT = 1;
    PREEMPTED_BY_SCHEDULER = 2;
    PLACEHOLDER_REPLACED = 3;
  }

  // The name of the partition the allocation belongs to
  string partitionName = 1;
  // The application the allocation belongs to
  string applicationID = 2;
  // The UUID of the allocation to release, if not set all allocations are released for
  // the applicationID
  string UUID = 3;
  // The termination type as described above
  TerminationType terminationType = 4;
  // human-readable message
  string message = 5;
}
```

TerminationType

The currently defined *TerminationType* values and specification of the side that generates (Sender) and the side that actions and confirms processing (Receiver):

Value	Sender	Receiver
STOPPED_BY_RM	shim	core
TIMEOUT *	core	shim
PREEMPTED_BY_SCHEDULER *	core	shim

* currently not handled by the shim, core or both

When the placeholder allocation gets released the *AllocationReleaseResponse* is used to communicate the release back from the core to the shim. The response contains an enumeration called *TerminationType* and a human readable message. For tracking and tracing purposes we should add a new *TerminationType* specifically for the placeholder replacement. The shim must take action based on the type and confirm the allocation release to the core.
It should provide enough detail so we do not have to re-use an already existing type or the human readable message. The human readable format can still be used to provide further detail on which new allocation replaced the placeholder.
Proposal is to add: *PLACEHOLDER_REPLACED*

Value	Sender	Receiver
PLACEHOLDER_REPLACED	core	shim

As part of the Scheduler Interface cleanup (YUNIKORN-486) the *TerminationType* should be extracted from the *AllocationRelease* and *AllocationaskRelease* message. It is an enumeration that can be shared between multiple objects. YUNIKORN-547 has been logged to handle this as it has an impact on the code outside of the scope of gang scheduling.

AllocationAskRelease Response and Request

The allocation ask release right now can only be triggered by the shim. In order for the core to perform the cleanup when the placeholder allocation times out, we need to make this a bidirectional message. Similarly to the Allocation we would rename the

AllocationAskReleaseRequest to *AllocationAskRelease*, so we can use this message in both directions:

```
message AllocationReleasesRequest {
...
  // Released allocationask(s), allocationasks can be released by either the RM or
  // scheduler. The TerminationType defines which side needs to act and process the
  // message.
  repeated AllocationAskRelease allocationAsksToRelease = 2;
}
```

Similar processing logic based on the *TerminationType* which is used for allocations should be used for ask releases. In patched message form that would look like:

```
message AllocationAskRelease {
  enum TerminationType {
    STOPPED_BY_RM = 0;
    TIMEOUT = 1;
    PREEMPTED_BY_SCHEDULER = 2;
    PLACEHOLDER_REPLACED = 3;
  }
...
  // The termination type as described above
  TerminationType terminationType = 4;
...
}
```

Confirmation behaviour of the action should be triggered on the type of termination received. The core will confirm the release to the shim of all types that originate in the shim and vice versa.
A confirmation or response uses the same *TerminationType* as was set in the original message.

Scheduler storage object changes

AllocationAsk

In line with the changes for the communication the objects in the scheduler also need to be modified to persist some of the detail communicated. The AllocationAsk that is used in the communication has an equivalent object inside the scheduler with the same name. This object needs to be able to store the new fields proposed above.

Proposed new fields: *taskGroupName* and *placeholder*.

In the current interface specification a field called *executionTimeoutMilliSeconds* is defined. This is currently not mapped to the object inside the scheduler and should be added. Time or Duration are stored as native go objects and do not include a size specifier.

Proposed new field: *execTimeout*

Allocation

After the allocation is made an Allocation object is created in the core to track the real allocation. This Allocation object is directly linked to the application and should show that the allocation is a placeholder and for which task group. This detail is needed to also enable the correct display of the resources used in the web UI.
The propagation of the placeholder information could be achieved indirectly as the allocation object references an AllocationAsk. This would require a lookup of the AllocationAsk to assess the type of allocation. We could also opt to propagate the data into the Allocation object itself. This would remove the lookup and allow us to directly filter allocations based on the type and or task group information.
From a scheduling and scheduler logic perspective the indirect reference is not really desirable due to the overhead of the lookups required. This means that the same fields added in the AllocationAsk are also added to the Allocation object.

Proposed new fields: *taskGroupName* and *placeholder*.

To support the release of the allocation being triggered from the core tracking of the release action is required. The release is not final until the shim has confirmed that release. However during that time period the allocation may not be released again.

Proposed new field: *released*

At the point that we replace the placeholder with a real allocation we need to release an existing placeholder. The Allocation object allows us to specify a list of Allocations to release. This field was added earlier to support preemption. This same field will be reused for the placeholder release.

Application

The AddApplicationRequest has a new field added that needs to be persisted in the object inside the scheduler.

Proposed new field: *placeholderAsk*

In the current interface specification a field called *executionTimeoutMilliSeconds* is defined. This is currently not mapped to the object inside the scheduler and should be added. Time or Duration are stored as native go objects and do not include a size specifier.

Proposed new field: *execTimeout*

The application object should be able to track the placeholder allocations separately from the real allocations. The split of the allocation types on the application will allow us to show the proper state in the web UI.

Proposed new field: *allocatedPlaceholder*

Queue & Node

No changes at this point. The placeholder allocations should be counted as “real” allocations on the Queue and Node. By counting the placeholder as normal the quota for the queue is enforced as expected. The Node object needs to also show normal usage to prevent interactions with the autoscaler.