Affinity and anti-affinity in Poseidon

-- draft document --

This document discusses how constraints can be modelled in Firmament. It provides solutions to the following issues:

f) Long running services may have multiple constraints themselves, for example, there may have affinity/anti-affinity between different components in a complete service. Currently, Firmament Cost model cannot represent the hard constraints and the constraints between tasks. We would like to explore solution to this regards. Ideally we would like to deploy components with affinity/anti-affinity constraints in one scheduling round.

h) Handling of complex affinity and anti-affinity inter-pod relationships.

1. Introduction

In this design document, I first give a bit of background knowledge on Firmament. Following, I explain how the affinity and anti-affinity constraints supported in Fleet and Kubernetes can be supported and implemented in Firmament/Poseidon (see https://coreos.com/fleet/docs/latest/affinity.html). Next, I differentiate between Kubernetes’ support for complex constraints and Poseidon/Firmament and explain how Poseidon could make better scheduling decisions. Finally, I include a list of high-level tasks that need to be finished for Poseidon to support affinity and anti-affinity.

In Firmament, each task (i.e., pod) has an associated TaskDescriptor (https://github.com/camsas/firmament/blob/master/src/base/task_desc.proto). Similarly, each resource (i.e., node or node sub-component) has an associated ResourceDescriptor (https://github.com/camsas/firmament/blob/master/src/base/resource_desc.proto). These

descriptors contain all the information related to tasks' requirements (e.g., start time, resource requests) and nodes' properties (e.g., resources used and reserved) respectively. However, they do not currently have a general mechanism of supporting user-provided metadata (e.g., Kubernetes labels). The first tasks we would have to do support general affinity and anti-affinity constraints is to extend the TaskDescriptor with support for label that are used to model requirements. Similarly, we would have to extend the ResourceDescriptor with support for labels that are going to be used to model node properties.

I now turn to describing how the various affinity and anti-affinity constraints offered by Fleet and Kubernetes can be modelled in Firmament's flow networks.

2. Affinity in Fleet

2.1. MachineID: Run a pod on a specific host.

This type of affinity is easily expressible in Firmament. I illustrate it with an example. In Figure 1, I show a graph with two tasks and two machines. T0 has a MachineID style affinity requirement for running on M0. In order to meet this requirement, we only connect the task to the node corresponding to machine M0 and to the unscheduled aggregator node. In this way, the task can either be scheduled on M0 or wait until M0 becomes available. This type of affinity can be obtained by doing the following adjustments to a cost model:
1) Make sure that GetTaskPreferencesArcs (https://github.com/camsas/firmament/blob/master/src/scheduling/flow/octopus_cost_model.cc#L139) only returns the machine on which the task must run.
2) Make sure that the task is not connected to any equivalence classes (https://github.com/camsas/firmament/blob/master/src/scheduling/flow/octopus_cost_model.cc#L113), because otherwise its flow could be sent along those arcs.

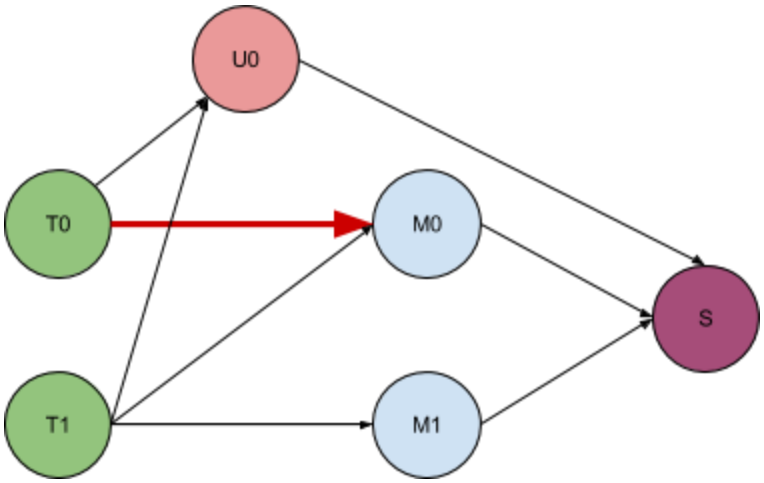

Figure 1: Example of MachineID constraint. T_0 is only connected to M_0 because it must run on M_0

2.2. MachineMetadata: Run on a host matching some arbitrary metadata.

This type of constraint is similar to the previous one. It can be handled in the same way in Firmament, with the only difference being that in GetTaskPreferenceArcs we would return all machines that match the arbitrary metadata.

2.3. Conflicts: Prevent a pod from running on the same host as some other unit.

I illustrate how this type of constraint can be supported with a simple example with two tasks and two machines (see Figure 2). The tasks should not be placed on the same machine. Firmament can model this constraint by introducing an equivalence class node (EC) for the tasks that should not run on the same host. It connects both tasks to the EC node and the node to the machines on which the tasks can run. It also sets a maximum capacity of 1 on the arcs that connect the EC to the machine nodes. This ensures that no two tasks that are only connected to unscheduled aggregators and this EC can simultaneously run on a machine.

In order to support such constraints, we would have to modify a cost model such that tasks that must not run on the same host are only connected to an EC. The method GetTaskEquivClasses should return this EC. Moreover, we would have to make sure that EquivClassToResourceNode (https://github.com/camsas/firmament/blob/master/src/scheduling/flow/octopus_cost_model.cc#L95) only returns maximum capacities of 1 for arcs that connect this EC node to machine nodes.

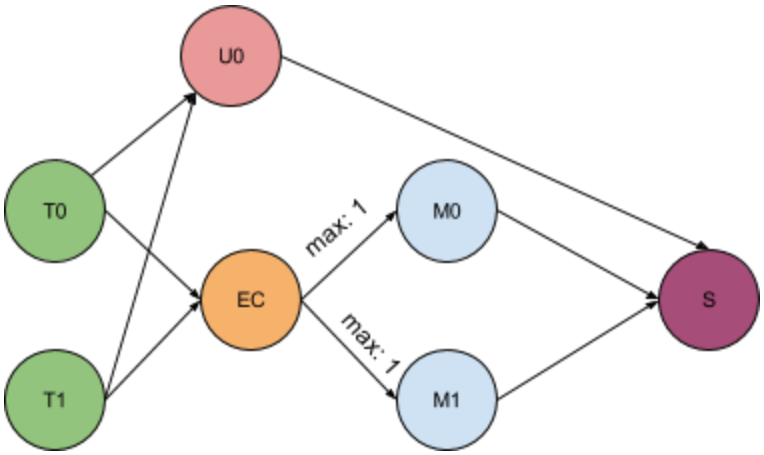

Figure 2: Example of Conflicts constraint

2.4. Global: Runs a pod on every node.

To support this constraint we would create as many task nodes as machine nodes exist in the graph (see Figure 3). Following, we would connect these task nodes to an equivalence class node (EC). We would then connect this equivalence class node to all the machine nodes. Like

previously, we would set a maximum capacity of 1 on these arcs to make sure that no more than one of these tasks executes on a machine. Moreover, if we want for all tasks to be guaranteed to execute at the same time no matter what then we also set a minimum flow requirement of one on the arcs. A minimum flow arc requirement is a constraint in the flow network. An arc with a minimum flow requirement of f, must route at least f units of flow for the flow to be a feasible solution.

In addition to the changes required to support conflicts, we would also have to change several methods in Firmament's cost_model_interface.h (https://github.com/camsas/firmament/blob/master/src/scheduling/flow/cost_model_interface.h) to return tuples of (arc cost, arc minimum flow requirement, arc maximum capacity) rather than just (arc cost, arc maximum capacity). Our min-cost flow solver (Flowlessly) supports minimum flow requirements, but it's an untested feature that may require debugging.

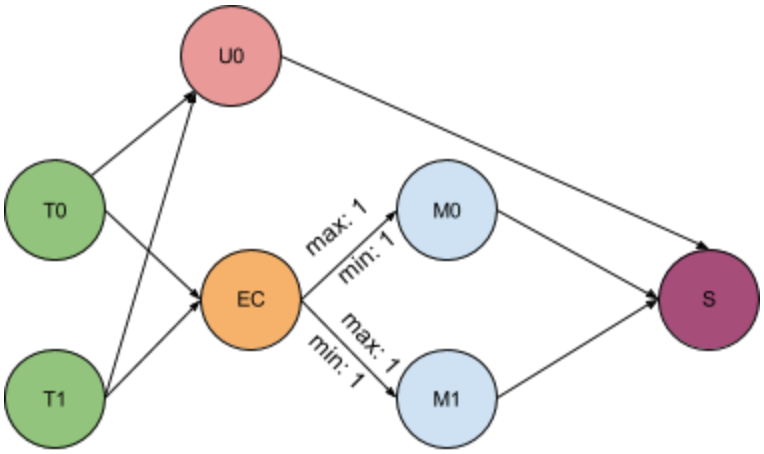

Figure 3: Example of Global constraint

2.5. Replaces: Runs a pod instead of another pod.

I illustrate this type of constraint with a simple example (see Figure 4). In this example, we have task T0 that is running on machine M0.

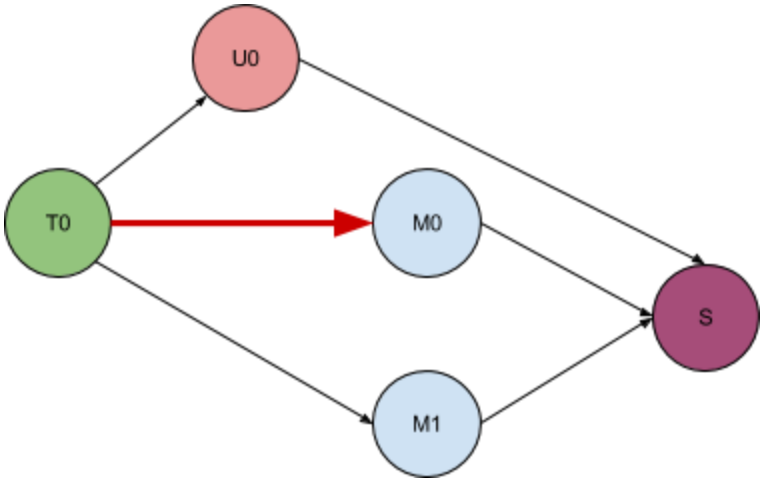

Figure 4: Example of Replaces constraint (before replacement task has been submitted)

Following, a new task (T1) is submitted (see Figure 5). This task has a requirement to replace task T0 on machine M0. This requirement can be modelled in the flow network in the following way: 1) remove all T0's arcs except the one connecting it to the unscheduled aggregator node, and 2) connect T1 only to the node on which T0 currently runs. Like, previous constraints, this type of constraint can be obtained by making sure that GetTaskPreferenceArcs returns M0 for task 1 and nothing for T0.

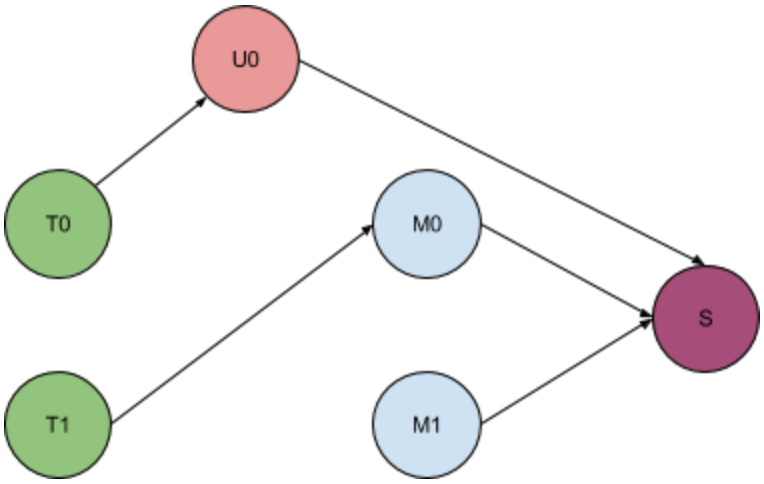

Figure 5: Example of Replaces constraint (after replacement task has been submitted)

2.6. MachineOf: Run a pod on the same host as another pod.
I discuss this type of constraint in Section 3.2.

3. Affinity in Kubernetes

3.1. nodeSelector: Given a set of (key, value) requirements, a pod can be scheduled to run (or not run) on certain nodes that meet the (key, value) requirements.

This type of constraint is similar to the arbitrary metadata constraint from Section 2. It can be implemented in the same way in Firmament.

3.2. Affinity

There are several types of affinity selectors supported in Kubernetes (see https://coreos.com/fleet/docs/latest/affinity.html#affinity). These types of affinity constraints can be grouped into three categories: soft, hard and complex constraints. I now turn to describing how these types of constraints can be modelled in Firmament's flow network.

3.2.1. Soft constraints

Some tasks may have a preference for several requirements to be met, but can also run if the requirements are not met. These constraints are called soft constraints and are specified in Kubernetes with the help of preferredDuringSchedulingIgnoredDuringExecution affinity selector.

Firmament is well suited to handle such constraints. Firmament's cost models add direct arcs from task nodes to machine nodes that satisfy the constraints. However, they can also add direct arcs to machine nodes that do not satisfy the constraints. The cost models ensure that the cost on the arcs that connect tasks to machines that satisfy the constraints are smaller than the costs on the arcs that connect tasks to machine that do not satisfy the constraints. Hence, the min-cost flow solver is incentivized to place tasks on the machines which satisfy the constraints.

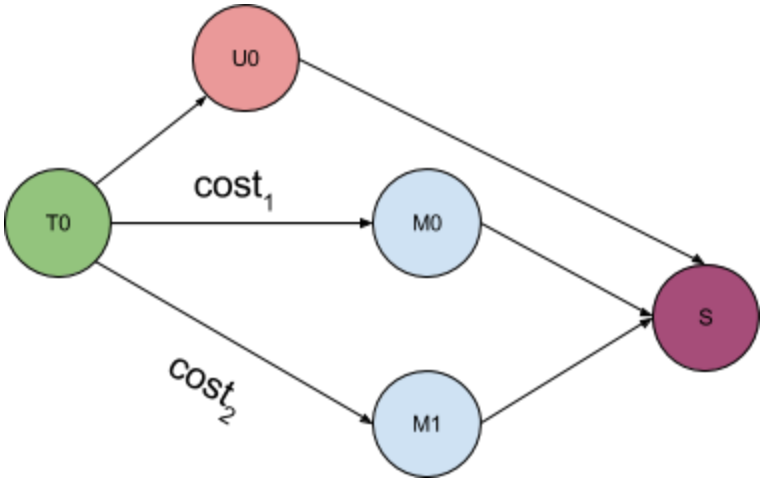

Figure 5: Example of a soft constraints.
T0 prefers machine M0. Thus, the cost model sets $cost_1 < cost_2$

3.2.1. Hard constraints

Some tasks can only be placed on machines that meet certain requirements. These requirements are called hard constraints. In Kubernetes, hard constraints are expressed using the requiredDuringSchedulingIgnoredDuringExecution and the requiredDuringSchedulingRequiredDuringExecution affinity selectors.

These constraints are similar to nodeSelector affinity and can be implemented in the same way: a node representing a task with hard constraints is only connected to an unscheduled aggregator node and to nodes corresponding to machines that satisfy the constraints. As a result, the task can either schedule on one of these machines or remain unscheduled until machines that meet the constraints become available. In Figure 6, I illustrate hard constraints with an example in which task T_0 and task T_1 are only connected to machines which satisfy their constraints.

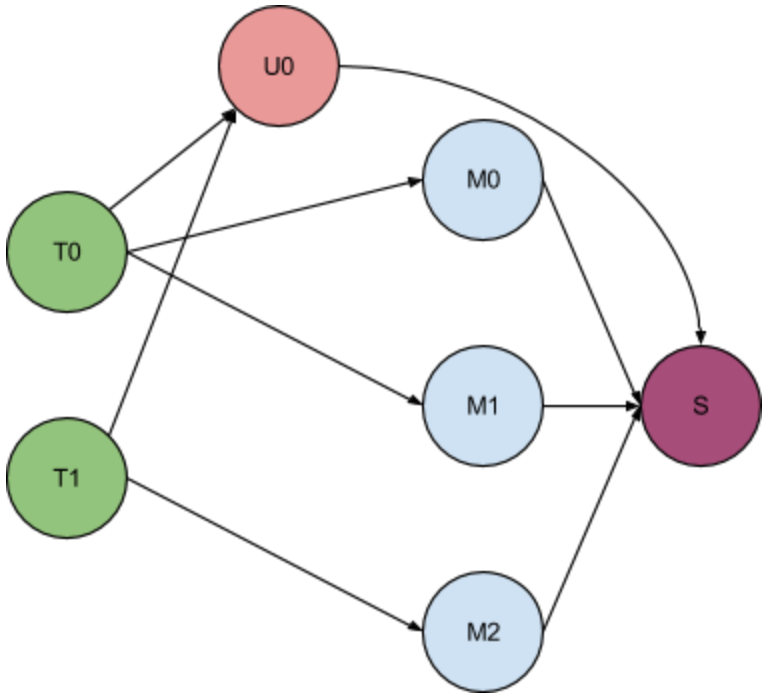

Figure 6: Example of a hard constraints. M_0 and M_1 satisfy T_0's constraints and M_2 satisfies T_1's constraints.

Firmament's advantage over queue-based schedulers (e.g., Kubernetes and Docker Swarm schedulers)

Queue-based schedulers place tasks one-by-one. As a result, they cannot consider how a task's placement affects placement options of other unscheduled tasks. Consider, for example, a scenario in which a cluster has available only a single machine with a GPU. In this cluster, there two machine learning tasks among many other tasks waiting to be placed. In a Kuberenetes-style queue-based scheduler, one machine learning task is before the other task in the scheduler queue. This task has a preference for running on the machine with the GPU. The other task is behind in the queue, but has a stronger preference than the first task for running on the same machine. A queue-based scheduler would place the first task on the machine with the GPU, to only be later faced with two sub-optimal options when placing the second task: (i) assign the second task to a sub-optimal machine or, (ii) migrate the first task to another machine – potentially losing all the work the task has done – and replace it with the second task. This is a

fundamental limitation of queue-based schedulers that increases task makespan or causes work to be wasted. By contrast, Firmament considers all unscheduled tasks at the same time together with their soft and hard constraints. Thus, it can avoid unnecessary task migrations and wasting task work.

3.2.3. Complex constraints

Complex constraints can be hard or soft in nature, and require several tasks and machines to simultaneously satisfy them. Task affinity and task anti-affinity are two popular types of complex constraints. Task affinity constraints are used to model requirements for placing two or more dependent tasks on a machine (e.g., a task that runs a web server and a task that runs a database used by the web server). By contrast, task anti-affinity constraints are used to model requirements for placing tasks on distinct resources. For example, some jobs may require tasks to be placed on different machines or racks for decreasing the likelihood of downtime in case of hardware failures.

Kubernetes supports a simple version of complex constraints in which a pod can be placed or not placed on a node depending on which other pods are or are not running on that node. Such constraints are supported by Kubernetes because it places pods one by one. It initially places the first pod on a node, and following, the Kubernetes scheduler just has to place or avoid placing pods on this machine. However, there are more difficult to satisfy complex constraints such as gang scheduling. Gang scheduling is an instance of complex placement constraints because several tasks have mutual dependencies: they all must be placed or none. Tasks/pods that have gang scheduling constraints can not progress unless all pods/tasks execute. Thus, schedulers that do not support gang scheduling constraints can waste resources with tasks that have been placed, but can not proceed because not all the tasks they communicate with are running. Queue-based schedulers struggle with such constraints because they can not be sure that one they place a task they will be able to also place of the remaining tasks that must be gang scheduled.

Firmament reconsiders all tasks in each scheduling round, but its current cost models do not generate flow networks that encode dependencies between tasks and machines. Tasks' flow supply is independently routed to the sink node. I now turn to describing two solutions for this limitation: 1) one based on load admission and multi-round scheduling, and 2) one that extends min cost flow-scheduling and introduces two new flow network constructs.

3.2.3.1. Solution based on load admission and multi-round scheduling

One way to support for the same type of complex constraints as Kubernetes does is to model it's queue-based approach in Firmament via reactive multi-round scheduling. With multi-round scheduling tasks with affinity constraints would not be placed in a single scheduling round. I first describe how the mechanism would work for tasks with affinity constraints, then I present how it

would work for tasks with anti-affinity constraints. Finally, I briefly describe this approach's limitations.

Task affinity: For every set of tasks with dependent affinity constraints, Firmament would first include only one of the tasks in the flow network. These per-set tasks would be encoded in the flow network without allowing for the possibility of them being migrated. Following, once such a task would be scheduled, Firmament would include in the flow network all the other tasks that have an affinity constraint with this task. These new tasks would only be connected to the machine node on which the task was placed. Thus, all the tasks would be co-located on the same machine. This approach can be generalized to other types of resources (e.g., racks, cores).

Task anti-affinity: Firmament would apply load admission and only introduce tasks with anti-affinity constraints one-by-one in the flow network. These tasks would not be allowed to migrate and they would not be connected or have paths to the machines on which other tasks with which they have anti-affinity constraints run. In this way, no two tasks with anti-affinity constraints would be co-located.

Limitations:
The solution based on load admission and reactive multi-round scheduling suffers from two main limitations:

1) Tasks can not be preempted or migrated once they are placed. This can increase task runtime or decrease task performance metrics because slow machines are used despite faster machines becoming available. Moreover, this could even cause priority inversion: high priority tasks may have to wait before running for low priority non-preemptable tasks with affinity/anti-affinity to finish.
2) Tasks could spend significant amount of time waiting to be scheduled. Especially, tasks with anti-affinity constraints, which would only be placed one by one. Firmament and min-cost flow-schedulers in general take longer to do a round of scheduling (in which they consider the entire workload) than it takes a queue-based scheduler to place a single task.

3.2.3.2. Solution based on flow network constructs

Complex constraints can be represented in flow networks using two flow constructs that I introduce now.

“xor” flow network construct

It is often desirable to express in the flow network exclusive disjunctions. These can be useful to express anti-affinity constraints. For example, avoid co-locating interfering tasks, or spread tasks across machines/racks to improve fault tolerance. In Figure 7, I show how a “xor” logical operation can be modelled in the flow network for n tasks’ flow supply. I connect the tasks that

to a new aggregator node (EC_G) I introduce. I then only connect this node to another new node (EC_O). I set a maximum capacity of p on this arc to ensure that only p units of flow supply can be routed along this arc. Moreover, I set a maximum capacity of r on the arcs that connect EC_O and the machine nodes. Thus, only at most p tasks can be placed at a time and not more than r tasks are placed on a machine. For p = 1 and any r > 0, the flow network construct is equivalent to xor.

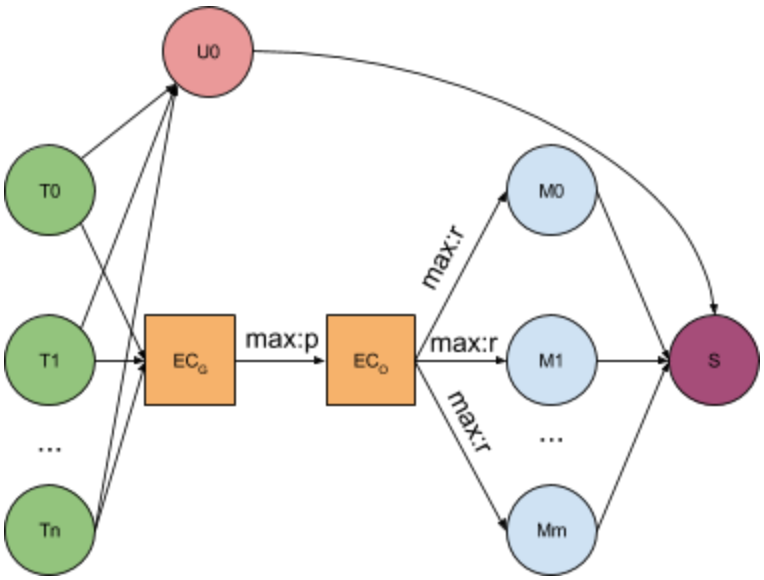

Figure 7: Example flow “or” construct

The “xor” network flow construct can be used to express complex conditional preemption constraints such as if task T_x is placed, then task T_y must be preempted as a result (i.e., Fleet’s replace constraint). Similarly, the generalised “xor” construct can be used to express conditional constraints, but also complex co-scheduling constraints (such as task anti-affinity constraints) that require several tasks to not share a resource (e.g., rack, machine).

In Figure 8, I show an example of how task anti-affinity constraints can be accomplished using the flow “or” network construct. In the example, there are three tasks that require to be placed on different machines. All tasks are connected to an equivalence class (EC_G) node which is in turn connected to another equivalence class (EC_O) node. EC_0 is also connected to the machines on which the tasks can run. However, the arcs that connect EC_0 to machines have a maximum capacity of 1 because tasks cannot be co-located on machines. The maximum capacity is a constraint that ensures that no more than one of these three tasks can be placed on each machine. Thus, the task anti-affinity requirement is satisfied.

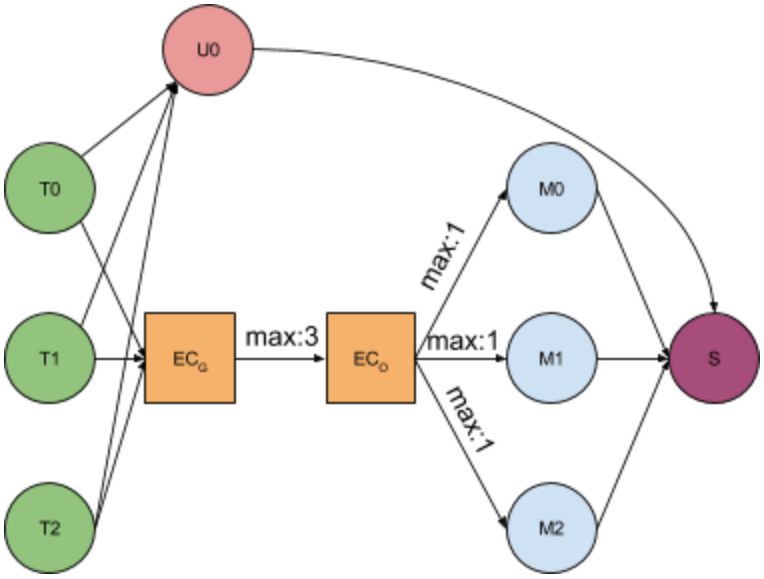

Figure 8: Anti-affinity for 3 tasks using the “xor” flow network construct

“and” flow network construct

Task affinity constraints express requirements that must be satisfied for two or more tasks. It is possible to build a flow network construct that forces tasks to satisfy such requirements. Similarly to the “xor” flow network construct, this construct contains an arc (EC_G, EC_E) along which tasks’ flow supply can be routed. But in contrast, I set a minimum flow requirement on this arc to enforce that all tasks’ flow supply is routed along the arc, and that all satisfy the requirements. However, this construct always routes flow along arc (EC_G, EC_E) which means that the tasks are always placed no matter how costly it is. This could sometimes cause low priority tasks to preempt or leave high priority tasks unscheduled. Unfortunately, it is not possible to address the limitations in min-cost flow networks, but it is possible in generalised min-cost flow networks.

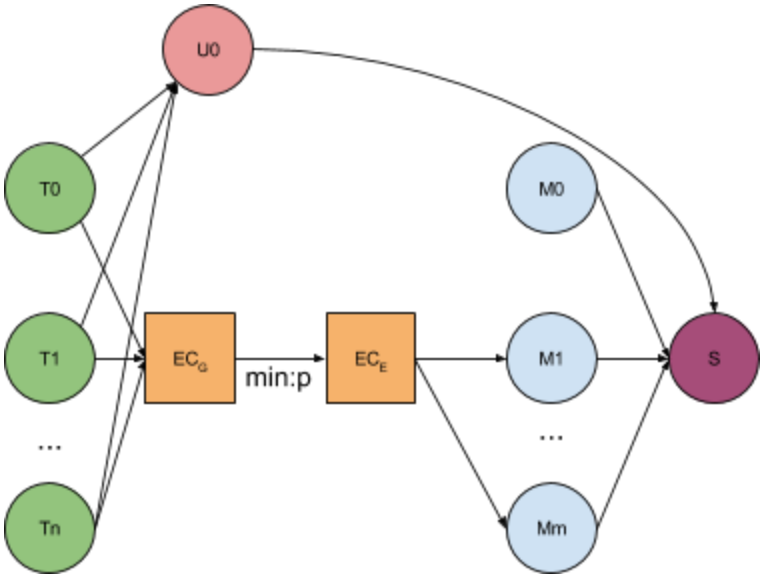

Figure 9: p tasks must be gang-scheduled on several machines that satisfy some requirements.

The generalised min-cost flow problem is a generalisation of the min-cost flow problem in which each network flow arc (i, j) also has associated with it a positive multiplier $\gamma_{i,j}$, called gain factor. In this network, for each unit of flow that enters arc (i, j) at node i, node j receives $\gamma_{i,j}$ units of flow. I use γ factors to build an "and" flow network construct that I show in Figure 10. Task T_o and T_1 must be either co-located or left unscheduled. I connect the task nodes to a new aggregator node (G), which I then connect to node E and node A. I set the minimum flow requirement and the maximum capacity of arc (G, E) to one, and I set a maximum capacity of one and a γ gain factor of two on arc (G, A). These arc constraints and requirements limit the possible ways of routing tasks' flow supply to two scenarios:

1. A task flow supply is routed via the unscheduled aggregator node U_o, while the others supply is routed via node aggregator G. From this node, the flow is routed to node E because arc (G, E) has a minimum flow requirement of one. Thus, both tasks remain unscheduled.
2. Both tasks' flow supply is routed via node aggregator G. From this node, exactly one unit of flow is routed to node E because arc (G, E) has minimum flow requirement and maximum capacity equal to one. The other unit of flow is routed along arc (G, E). Node A receives two units of flow because arc (G, E) has associated with it a γ factor of 2. Thus, two units of flow are routed to machine M 0 , which means that both tasks are placed.

The "and" flow network construct can be used to express complex task affinity constraints such as: "all tasks of a job must be placed on the same resource". In contrast to the previous solution, tasks can stay unscheduled if higher priority tasks must be placed.

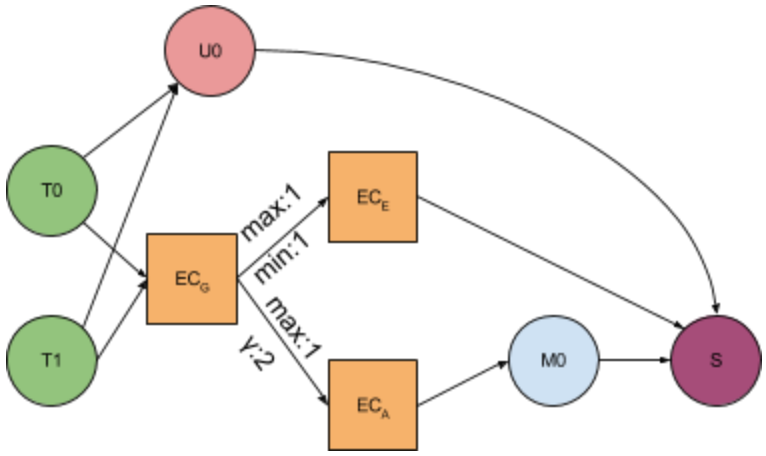

Figure 10: Example flow "and" construct

TODO(ionel): 1) Check if there is a simpler solution for the simple inter-pod affinity that Kubernetes currently supports (https://kubernetes.io/docs/user-guide/node-selection/)

TODO(ionel): 2) Check what happens with the excess flow in the generalized min-cost flow problem.

For a more in-depth explanation of these solutions you can check Section 5.4.2 of my PhD thesis draft (https://www.cl.cam.ac.uk/~icg27/pub/drafts/thesis.pdf).

5. Task list

This is an initial list of the tasks we would have to do to support task affinity and anti-affinity Poseidon/Firmament. The list may expand as we improve this support.

1. Extend TaskDescriptor and ResourceDescriptor with support for Kubernetes-style labels.
2. Populate TaskDescriptors/ResourceDescriptors with labels in Poseidon.
3. Adjust the methods that return pair<Cost_t, max capacity> in cost_model_interface.h to also return minimum flow requirement. Also adjust all the cost models that implement the interface.
4. Test/debug Flowlessly correctly handles minimum flow requirements.
5. Extend Firmament with support for arc gain factors. This requires extending the FlowGraphArc, CostModelInterface, all the cost models that implement it, the code that uses the interface, and the code that exports the graph to Flowlessly.
6. Implement a generalized min-cost flow algorithm in Flowlessly or adjust one of the existing algorithms. NOTE: This step may take a lot of time because it may be difficult to convert an existing algorithm to solve the generalized min-cost flow problem.
7. Implement task affinity and anti-affinity in a cost model.

6. Use case

Application that has n instances. Each instance should run in a different availability zone or in an availability zone in which not too many instances of the same app are running. Moreover, within the availability zone we want to choose the node with the most free resources.

These requirements can be satisfied by using a combination task anti-affinity constraints (explained in 3.2.3.2) and arc costs. I show how these can be used with a simple example. In Figure 11, I have 3 tasks that need to be placed in different availability zones. For each availability zone i, I introduce a special equivalence class node AV_i. I connect all the machines in each availability zone with the zone's corresponding equivalence class node (AV_i). I also introduce, an equivalence class (EC) to which I connect the three tasks. I also connect this EC node to the per availability zone equivalence class node. I set a maximum capacity equal to r on the the arcs (EC, AV_i). This maximum capacity enforces that no more than r tasks will be placed in an availability zone. For example, if we only want an instance of the app in an availability zone then we set r to 1.

FInally, in order to incentivize the solver to place the tasks on the node with the most free resources from an availability zone, I set the costs on the arcs connecting AV_i nodes to machine nodes to be proportional with the machine utilisation. For example, if $M_{1,0}$ has more free resources than $M_{1,1}$ then I set c_{10} to be smaller than c_{11}. Thus, the solver will prefer to push flow to $M_{1,0}$ rather than $M_{1,1}$.

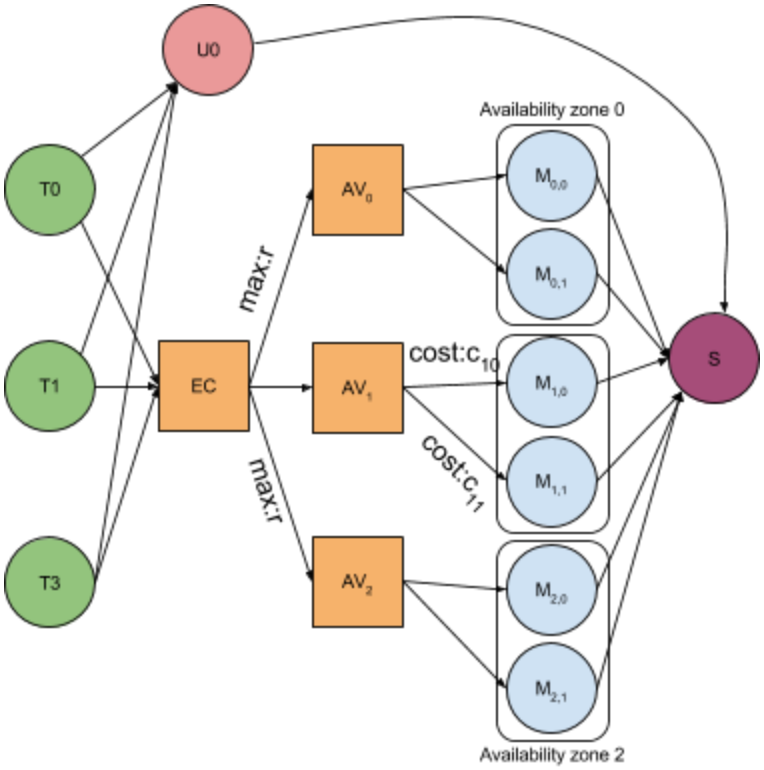

Figure 11: Example in which tasks are spread across availability zones.