Scheduling and Orchestration Discussion

Mike Spreitzer
September 16, 2013

Background:
Unified Resource Placement Module - https://docs.google.com/document/d/1cR3Fw9QPDVnqp4pMSusMwqNuB_6t-t_neFqgXA98-Ls
NovaSchedulerPerspective - https://docs.google.com/document/d/1_DRv7it_mwalEZzLy5WO92TJcummpmWL4NWsWf0UWiQ
Scheduler group chat Aug 13 - http://eavesdrop.openstack.org/meetings/scheduler/2013/scheduler.2013-08-13-15.01.html
Scheduler group chat Sep 3 - http://eavesdrop.openstack.org/meetings/scheduler/2013/scheduler.2013-09-03-15.02.html
Scheduler session planning for Icehouse - https://etherpad.openstack.org/IceHouse-Nova-Scheduler-Sessions
Heat Native DSL blueprint - https://blueprints.launchpad.net/heat/+spec/open-api-dsl
Early Proposal for HOT - https://wiki.openstack.org/wiki/Heat/DSL
Another Early Proposal for HOT - https://wiki.openstack.org/wiki/Heat/DSL2
Some Open Questions About HOT - https://etherpad.openstack.org/heat-dsl-questions

Following are some thoughts centered around scheduling and orchestration, based on: my group's experience with running code that does holistic infrastructure scheduling then orchestration, our thoughts about improvements to that code, and initial thoughts about how that sort of thing could be fitted into the OpenStack architecture. We work closely with a group that owns code that has an interesting approach to software coordination, they are interested in contributing it to OpenStack, and that is included in here too.

The OpenStack (nova) scheduler group has been discussing how to add holistic infrastructure scheduling. By holistic infrastructure scheduling I mean a scheduler that will look at a whole infrastructure template (AKA pattern AKA topology), including all the relevant types of resources, and make a joint placement decision. The Unified Resource Placement Module (u-rpm) proposal has orchestration downstream from holistic scheduling, as does my group's running code. By orchestration I mean the issuing of the calls on lower level APIs (with suitable ordering, and parallelism where possible) to create the infrastructure according to the original request and the scheduler's decisions.

When holistic infrastructure scheduling is being done, the input to that scheduling can usefully include additional kinds of policy and relationship information that is not defined in CFN templates. Examples include co-location and anti-co-location constraints. There are ways to include such information --- albeit not in a way that will have any effect on an ordinary CFN engine --- in valid CFN templates, taking advantage of certain places you can put things that the

CFN engine does not interpret. I call these “I-Specialized” CFN templates (“I” is for Infrastructure).

A peer group has a software coordination technology that involves no infrastructure-level dependencies between VMs. Rather, all the software coordination between VMs is handled by framework facilities as the VMs run their startup scripts (userdata). This framework uses a coordination service (ZooKeeper, in particular). There are ways to describe software that is to be coordinated this way in CFN templates, putting information in places that an ordinary CFN engine ignores. I call such an augmented CFN template “S-Specialized” (“S” is for Software).

We hope that HOT will eventually be a more natural description language.

See https://docs.google.com/drawings/d/1CZcrbhkwUDz5IZFdngbBaEitry25bt9OEpehn2n7skY for the big picture of how these pieces might fit together. A template that uses the aforementioned software coordination and infrastructure concepts can be run through a Software Coordination service that effects the early phases of the software coordination technology and produces a template that is usable by an engine that knows nothing of this software coordination technology (a colleague has demonstrated this), provided that the necessary coordination service is available to the VMs as they start up. That not-S-specialized but still I-specialized template is input to holistic scheduling, which makes a joint placement decision and writes it into a plain template (note that OpenStack allows the placement to be given by the Nova and Cinder clients (more or less; here we require it to be fully true)). The plain template is then the subject of infrastructure coordination.

See https://docs.google.com/drawings/d/1o2AcxO-qe2o1CE_g60v769hx9VNUvkUSaHBQITRl_8E for a picture that zooms in a little on the infrastructure part of the story. My group’s current running code does not use CFN templates at any stage. The picture shows our planned improved version that takes I-Specialized CFN templates (as the language for representing a VRT - Virtual Resource Topology) as input to the holistic scheduler. You see the story is a little more complicated than appears in the earlier outline. For brevity this picture focuses only on the sort of request that instantiates or revises a template (they are equivalent here: they assert the desired topology for a given stack, in Heat terms).

Desired topologies are logged in the VRT log. The holistic scheduler takes a whole topology at a time and can make a joint placement decision for all the resources in that topology. However, decisions that do not interact with other decisions can and should be delegated to isolated schedulers in lower level components. Once the decisions are made for a VRT, the agumented (by placement decisions) VRT is written into the target state. The target state logically contains a copy of the original requested topologies; this can be implemented by simply referencing the originals as they sit in the VRT log --- with suitable constraints on the pruning of that log. The holistic scheduler judges available capacity by subtracting existing allocations from raw capacity. The existing allocations that are subtracted come from the scheduler’s effective state, which is

the union of the target and observed states. By “union” I mean to not double count a virtual resource that appears in both states with the same placement.

The infrastructure orchestrator’s job is to use the lower level APIs to change the real state so that the observed state matches the target state, tracking both target and observed state as they change.

The observer’s job is to maintain a convenient copy of the real state, called the observed state. The observed state will, in general, lag that real state. The observed state, and its copy in effective state, are “soft state”; they can be lost at any time and then reconstructed from the real state. The observer can use any combination of polling and subscription to get its job done. Current lower level APIs do not have a way to subscribe to change notifications; that should change.