Local Scheduling

Greg Haynes (greg@greghaynes.net) Tara Gu (guw@ibm.com)

In order to remove the Kubernetes API server and control plane from Knative's dataplane we would like to own scheduling decisions for pods, at least in the cold-start case.

Activator Daemonset and Local Scheduling

We will run Activators as DaemonSets which schedule pods to the node they reside on. We can add an environment variable FieldRef of the nodename for the node each activator daemonset is running on. Using this, we can have each activator write pod definitions to kube-api with nodeName pre-populated to the same node that activator runs on. Although this will still make use of Kube-api, kubelet is the only controller which is in the critical path of these resources being realized and because kubelet is scaled out to each node it has much better performance characteristics.

Note that the local scheduling only applies for the use case of scaling from 0 to N. Although it may not be in scope of the initial implementation, we can kill the locally scheduled pods once we have scaled to N.

Node Failures:

We need to maintain a two-way mapping between a set of nodes and their revision endpoints. When a node dies, we need to reverse look up for the corresponding revisions, and update the endpoints of those revisions to remove the failed node.

We also need to implement retrying for the case when a request is sent to a failed node. It is possible a request will be forwarded to a dead node before we are able to remove it. In this case we need to make sure the dataplane is able to retry the request to an alternative node.

Recovery from a node failure will be handled the same way as today, i.e. the current replica count doesn't match the desired replica count, and therefore the deployment controller will create new pods.

Node Fullness:

We also need to rely on retrying in the case of many requests being sent to one node. Note that the local scheduling applies for the use case of scaling from 0 to N. Currently activators don't scale linearly with the number of nodes. Even so, the number of writes from activator to autoscaler is largely a function of request count, not activator count. As a result we do not expect much performance impact from this change. We can validate this by creating a load test the current implementation and compare it with the performance with local scheduling.

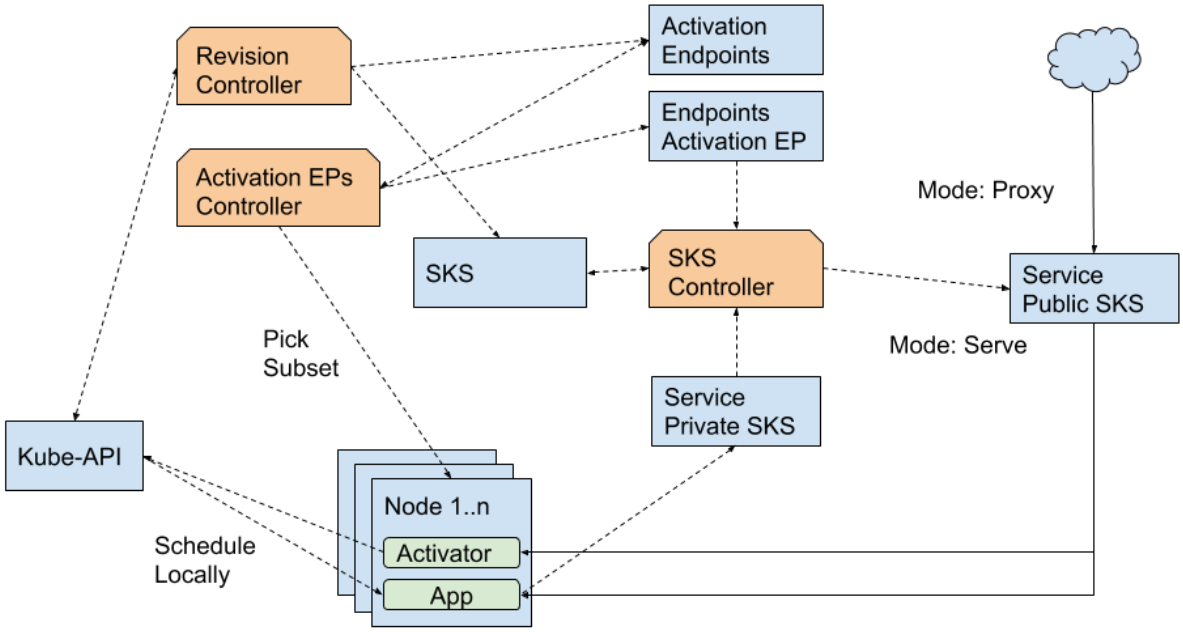
Race Condition:

One thing to be aware of is a race condition when you try to add a pod to a deployment and increase its replica count. If the pod is created first the deployment will see one too many pods and try to delete one. When the replica count is increased first it will try to create a pod to match the new replica count and there will be one too many pods. Creating too many pods is probably the better choice there. Note that this only applies in the scale from zero scenario.

Maxscale:
It is possible for a revision to have maxscale < |revision activators|. In this case if every revision activator gets a request while the revision is in proxy mode we need to avoid creating more than maxscale pods for that revision. We can do this by having activators check replicaCnt of the revision's deployment before scheduling locally, if replicaCnt >= maxscale then the activator will forward a request using the normal forwarding logic (revision endpoints).

Activator Endpoint Subsetting

If we simply run daemonsets as described above and schedule locally then applications will pay cold start cost whenever their requests lands on a different node (which, for a large cluster, can be a significant number of times). We can perform ahead-of-time scheduling by using our (already manually controlled) public SKS and filling it with a subset of the activator pods. For example, if an application's SKS is in proxy mode and we fill its public endpoint with 3 activator pods then at most cold start will be paid 3 times regardless of cluster size.



Activation Endpoints

We will introduce a new CRD named ActivationEndpoints. These will be 1:1 with revisions and have an OwnerReference to the corresponding revision. We will also introduce a corresponding ActivationEndpointsController which watches for these resources and maintains an endpoint ({revision-name}-activationendpoints) consisting of the activator subset for that revision. Its role is to implement the subsetting algorithm which selects activators (and consequently nodes) for a revision and stores the result in the activator endpoint for that revision. It is implemented as a separate controller so consumers of Knative Serving can implement custom algorithms by replacing this controller.

Note that there may be a need to implement two way mapping between revision and activation endpoints, for the use case of updating all affected endpoints in the case of node failures.

Endpoint Subsetting Algorithm

Knative serving should supply a default / upstream implementation of the subsetting algorithm. All subsetting algorithms should take several properties into account:

- Applications can experience cold-start as many times as there are endpoints
- 1 endpoint is a SPOF for a revision

- This endpoint list will not reflect resource usage in near real time and therefore the availability of endpoints may be low. This can be counteracted with multiple endpoints.
- Load spread evenly across the cluster

For an initial implementation we can pick N random activators to use as an endpoint (where N = min(num_activators, 3)). To improve this we can keep track of how many revisions are scheduled to each activator and place revisions use the “least loaded” activators.

Choice of N:
Having N random activators as endpoints helps us to have a starting point of enough capacity until we are able to scale up via our autoscaler (because local scheduling only applies for the use case of initial scaling from 0 to N). We can use TBC, which indicates how much burst capacity we should support for a revision, to determine N:

If TBC != -1, we can make N a function of TBC and container concurrency: N = min(1, TBC - container concurrency). N should also be greater than 1 for fault recovery.

If TBC == -1, we need users to provide a config value `maxNodeSelection` as N with limits [1, number of nodes). We also need to indicate that there is not going to be high availability if N == 1.

Note that since the subsetting algorithm is easily replaceable, downstream can supply their own algorithm for choosing an N.

- Let’s not forget about custom schedulers or normal kube scheduling options - like taints. We advertise that this is leveraging Kube under the covers so if people can’t use their normal k8s plugins (schedulers) or options (taints) then it might not be “leveraging kube” as we say