Custom Kubernetes Scheduler

### Introduction

Kubernetes is a container management technology that is used to manage containerized applications (docker images of applications). It is an open source system which helps in creating and managing containerization of application that helps in moving from host-centric infrastructure to container-centric infrastructure [1].

#### Kubernetes components:

Programs running on Kubernetes are packaged as containers. Multiple programs can be added into a single container. But it is often better to have many small containers than a large container. A cluster deploys containerized applications without tying them to individual machines and Kubernetes automates the distribution and scheduling of application containers across a cluster in a more efficient way. A cluster contains two resources Master and nodes. Master is considered as a brain, it coordinates the cluster and automatically handles scheduling of pods across the Nodes in the cluster [2]. Master runs four services: Scheduler, kube-API Server, Controller Manager, etcd. Scheduler watches the pods and chooses the best fit of which nodes to run the pods on. Kube-API Server is the main management point of the Kubernetes cluster. It processes REST operations, serves the kubernetes API and is the only component to update etcd. Controller Manager runs the controllers in background which runs different tasks in kubernetes cluster. The various controllers it includes are Replication Controller, Endpoint Controller, Namespace Controller. It also does lifecycle functions like garbage collection. Kubernetes uses etcd to store metadata about the cluster in key/value database, which is widely distributed and is consistent. This metadata is used for service discovery, configuration management and coordinating distributed work.

#### Nodes:

Node is the smallest unit of computing hardware in kubernetes. It can either be a cloud or physical or virtual machine. It is a representation of a single machine in a cluster. Nodes run two services kubelet and kube proxy. Kubelet is also known as a pod agent, responsible for Kubernetes Master and Node communication. It manages pods on node,volumes, and creating new containers. Kube-proxy is a network proxy that runs on each node in the cluster. It is responsible for request forwarding and allows TCP and UDP stream forwarding [3].

#### 

#### Pods:

A pod is a collection of one or more containers inside a node of a Kubernetes cluster. Any containers in the same pod share the same resources, IP and hostname. Containers communicate with each other using the pod IP address. Pods are used as the unit of replication in Kubernetes. There are multiple copies of a pod running at any time in a production system to allow load balancing and failure resistance. With horizontal pod auto scaling , pods of deployment can be automatically started and halted based on cpu usage.

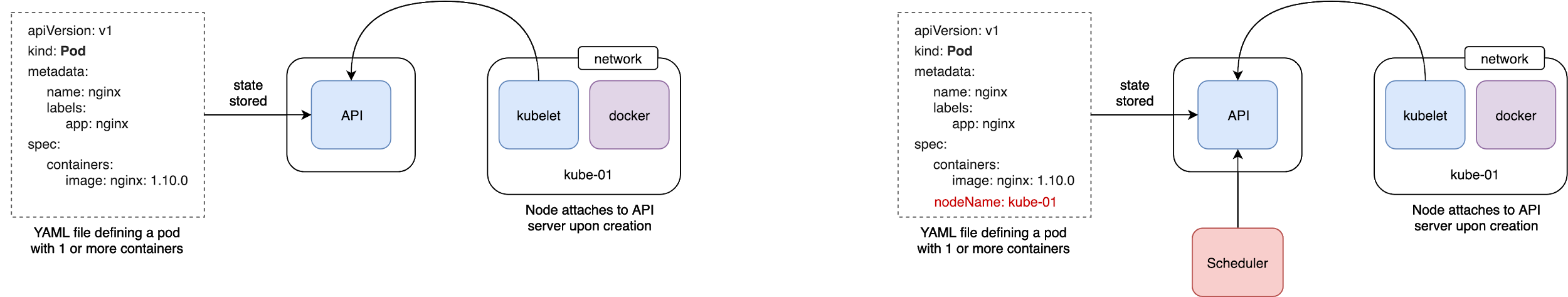
Pods are managed by deployment. A deployment’s primary purpose is to declare the number of replicas of a pod to be run at a time. It will create a pod by it’s specifications from the template. Their target is to keep the pods running and update them in a more controlled way. If a pod dies, the deployment controller replaces the instance with an instance on another node in the cluster automatically and provides a self-healing mechanism to address machine failure or maintenance. A service is responsible for making pods discoverable inside the network or exposing them to the internet and accessing it (virtual IP or port) without NAT. It identifies pods by its LabelSelector. The incoming requests are load balanced to the right pods.

#### Project goal:

The goal of the project is to build custom Kubernetes scheduler which optimizes for a different objective than the default scheduler, and to create a workload to demonstrate that our custom scheduler performs differently than the default scheduler.

### The default scheduler

When a pod is created, it does not actually start running. The desired state is simply stored in the Kubernetes API server. The scheduler is responsible for assigning the pod to a compatible node. Then, kubelet, an agent that runs on every node in the cluster, executes the assigned pod, i.e. runs all the containers in the pod.

[](https://www.draw.io/?page-id=Xm5XKVHC1VUXypb-YWJG&scale=auto#G18Mygy_30S9jXOy9FgLQgltUDDVK5Q-pG)

The default scheduler determines the placement using some constraints and chooses the ones with highest score. It first applies predicates, which will remove nodes that don't meet the requirements for the pod or don't have enough resources available [10]. The available resources for a node are determined by adding the resource requests (including CPU and memory) of all pods running on the node and dividing by the capacity of the node [10].

Then, the scheduler applies priority functions to determine the best node to schedule the pod on. The goal is to distribute the pods across different nodes and zones, and still favor nodes that are least loaded (based on the resources available). Additional factors, such as reducing the number of pods from the same service running on the same node, are considered as well [10]. The node with the highest priority (or a node tied for the highest priority) is chosen [9].

### Our custom scheduler

The objective of our custom scheduler is to optimize memory utilization across the nodes. The default scheduler considers cpu and memory requests [10] (which are optional in the yaml files specifying pods or deployments of pods) [8], while our scheduler focuses on memory usage on the nodes. It's possible for pods to use much more memory than requested, or to use much less memory than requested, or to not specify a request. After the pods are scheduled, they may not use the estimated amount of memory, so focusing on memory requests may not accurately reflect how much memory will be used.

Our custom scheduler makes decisions based on how much memory is available on the nodes, to address the case where the actual usage is very different from the requested usage. We use a metric (node\_memory\_memAvailable) to determine which node has the most memory available. With the default Kubernetes scheduler, if one node is running a pod that requested a lot of memory but was not actually using very much memory, and another node was running a pod that didn't have a memory request but was actually using a lot of memory, Kubernetes would likely schedule a new pod on the node that was using a lot of memory. Based on requests, it appears to be the least-loaded node. Our custom scheduler could spread out the memory usage across the nodes, instead of ending up with memory-intensive applications running on the same node.

Methodology

#### Approach:

The Kubernetes scheduler needs to do three main tasks: First, constantly monitor pods to find any unscheduled ones. Second, for each unscheduled pod, find the ‘best’ node. ‘Best’, in the context of our custom scheduler, means a compatible node with the most available memory. Third, bind the pod to the selected node.

Node selection (the second task) is the one that we have customized for our purposes - to favor the one with most available memory. This involves the following steps:

1. Find all compatible nodes that ‘fit’ the pod i.e. satisfy the pod’s requested CPU and memory requirements.
2. Retrieve the relevant metric data (available memory) for each compatible node.
3. Select the node with the optimum (maximum) metric value.

#### Tech stack:

We used the following tools to build and test our custom scheduler:

* Kubernetes + Docker
* Kubeadm-dind-cluster (KDC): to create a local multi-node cluster made of Docker containers
* YAML files: to define pods and deployments
* Prometheus Node Exporter: to scrape node metrics
* GoLang: custom scheduler development
* PromQL + Prometheus HTTP API: to query metric data

#### Metric Tool:

We have used Prometheus [5], an open source, systems monitoring toolkit, specifically the Node Exporter tool, to scrape metric data for all active nodes. Node exporter offers 43 metrics that expose memory information. “The amount of available memory on a Linux system is not just the reported “free” memory metric. Unix systems rely heavily on memory that is not in use by applications to share code (buffers) and to cache disk pages (cached). So one measure of available memory is: sum(*node\_memory\_MemFree* + *node\_memory\_Cached* + *node\_memory\_Buffers*)” [7]. Newer Linux kernels (after 3.14) expose a better free memory metric, *node\_memory\_MemAvailable*, which is the one we have chosen to use.

Note: “Prometheus Node Exporter is for \*nix systems, there is a WMI exporter for Windows that serves an analogous purpose” [6].

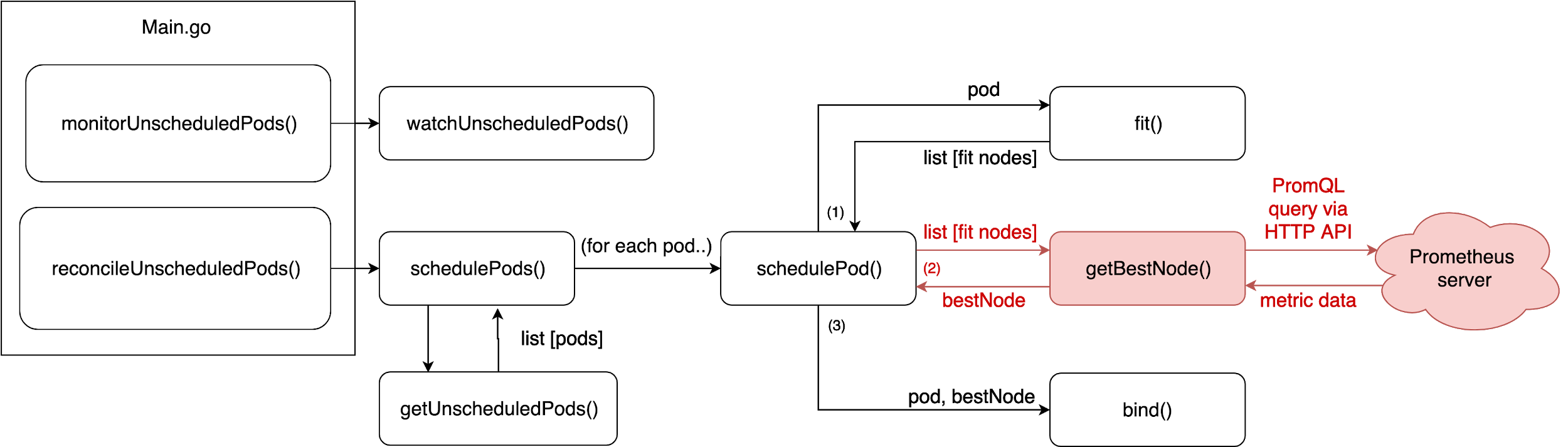
#### Implementation:

Our implementation was guided by Kelsey Hightower’s demo for building a custom Kubernetes scheduler [12]. Hightower demonstrates building a toy custom scheduler that schedules based on some manually added random annotations (each node is annotated with some random cost; and the scheduler picks the least cost node). We have used parts of his source code [13] as the base for our scheduler, specifically, the components for monitoring and finding unscheduled pods, running the default predicate checks to find nodes that satisfy the pods’ requested CPU and memory requirements, and binding the pods to the selected nodes.

Upon this base, we built components to factor in the node metric values from Prometheus, into the scheduling decision. Once the default predicate checks are run to find the list of compatible nodes that satisfy the pod’s requested CPU and memory requirements, a PromQL query is made via Prometheus’ HTTP API to get the *node\_memory\_memAvailable* metric values for the compatible nodes:

**resp, err := http.Get("http://localhost:8080/api/v1/query?query=node\_memory\_MemAvailable")**

Then, the node with the maximum value is identified and assigned to the pod. The following figure outlines the major components of the custom scheduler.

[](https://www.draw.io/?page-id=ZNf8Fa9ABcCM0Wy7kyj-&scale=auto#G1Ij7cNgJyL7LRL97kJ3m05Wwp8c3WO6G6)

### Testing

To test the custom scheduler against the default kubernetes scheduler, we deploy five pods:

Sleep(2 replicas), Nginx(1 replica) on each scheduler (default and the custom scheduler) and sysbench(1 replica).

The sleep.yaml is a pod that requests for 1800Mi of memory and sleeps infinitely. We deploy two pods of sleep, requesting for 3600Mi of memory in total. The testdefault.yaml runs the Nginx application on the default scheduler and testcustom.yaml runs the Nginx application on the custom scheduler.

The sysbench.yaml runs the sysbench workload for the memory test. The benchmark application allocates a buffer of size memory-block-size and then reads/writes from the buffer until a specific volume (memory-total-size) is reached. The number of threads and the type of operation(read or write, sequential or random) can be specified by the user[4].

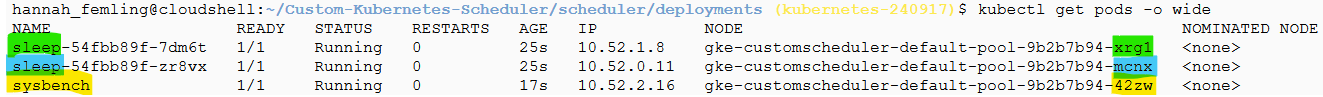
In the current project, we have allocated a buffer size of 2GB and the total volume to be read is 5TB. The read operation is performed in random by 10 threads.

In the beginning, the two pods running the sleep application and one pod running the sysbench workload are deployed. The three pods are deployed on the three different nodes. The fourth pod is deployed using custom scheduler (testcustom.yaml) and default scheduler (testdefault.yaml). The pods are deployed on nodes based on the metrics in the custom scheduler and the default scheduler. The results of the experiments performed to schedule the pods using the custom and the default scheduler are discussed in the next section.

Results

#### Setting up the initial state:

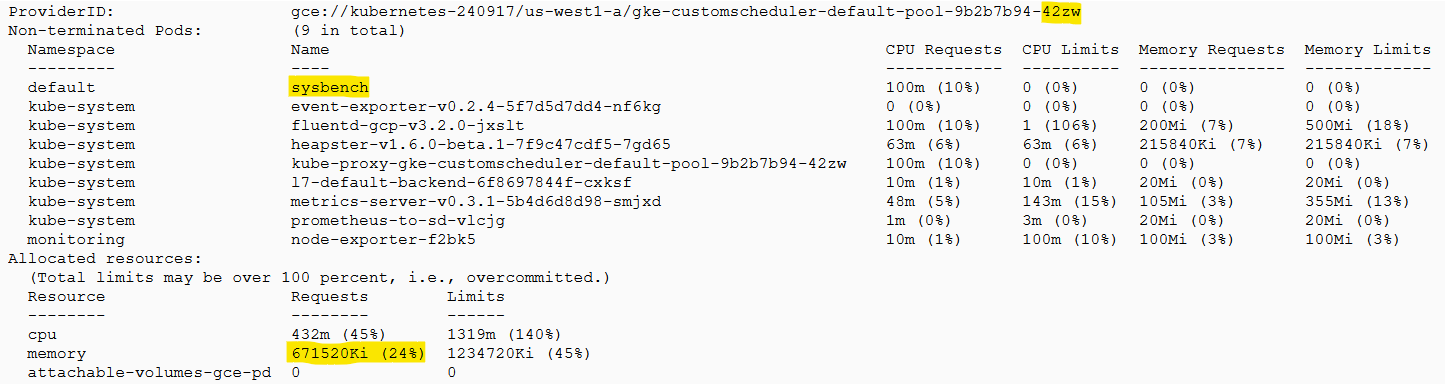
Initial pods running on the nodes, with sleep pods running on two nodes, and the sysbench pod running on the third node:



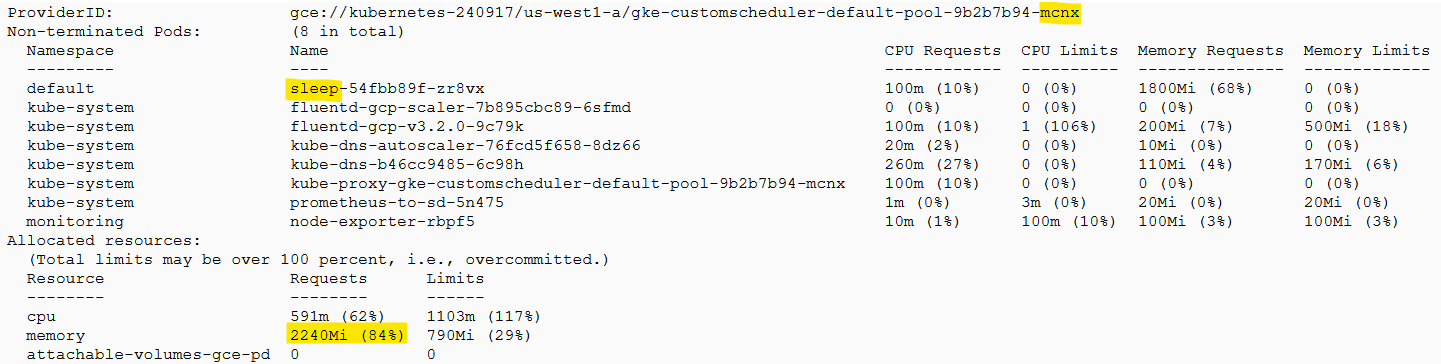
##### 

##### Initial memory requests for the nodes, shown by the output from the kubectl describe nodes command:

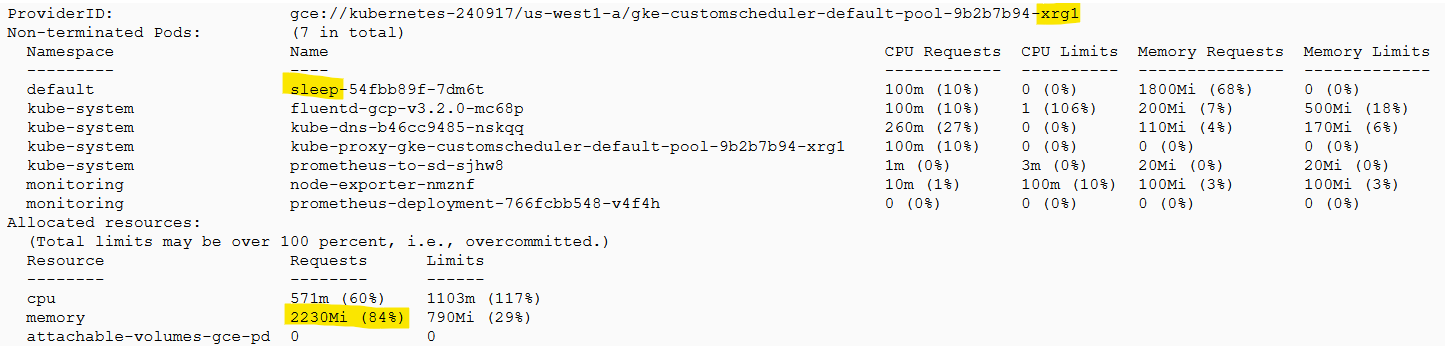
Node 42zw (running sysbench) has 24% of memory requested:



Node mcnx (running sleep) has 84% of memory requested:

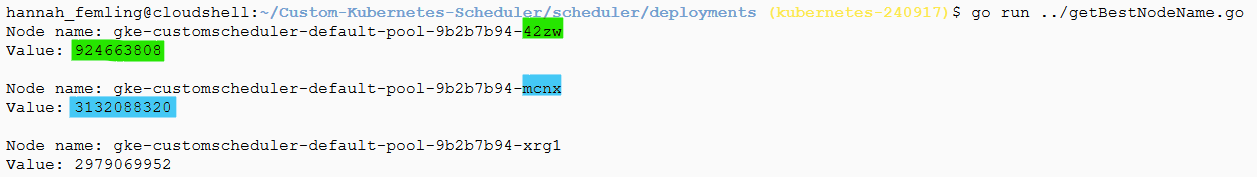


Node xrg1 (running sleep) has 84% of memory requested:



##### Initial memory utilization:

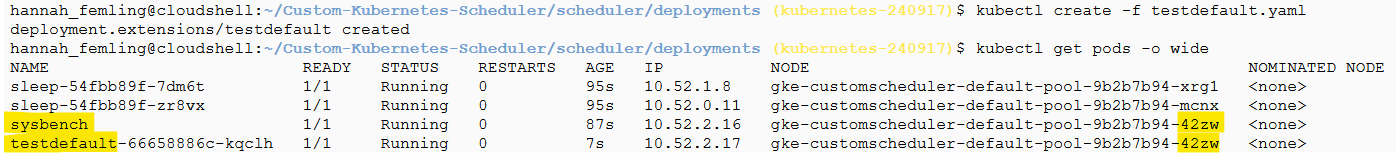
The initial node\_memory\_memAvailable metrics for each node:



The node 42zw, running the sysbench pod, has the least memory available. The node mcnx, running a sleep pod, has the most memory available.

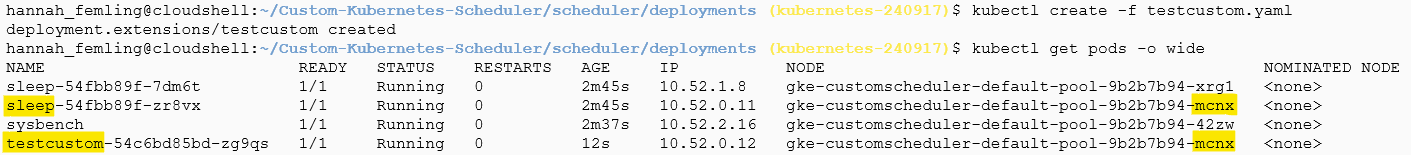
#### Scheduling a pod with the default scheduler:

When a new pod is scheduled with the default scheduler, it is scheduled on the node running sysbench, which is using the most memory:

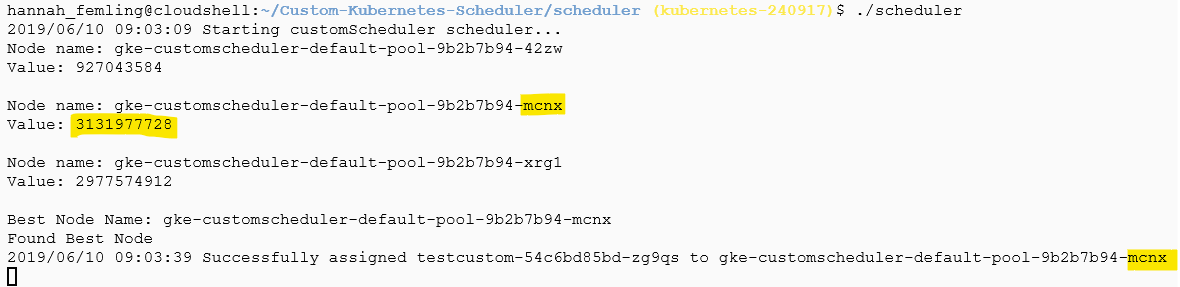


#### Scheduling a pod with our custom scheduler:

When a new pod is scheduled with the custom scheduler, it is scheduled on the node using the least memory:



The log from our custom scheduler, showing the metric values captured:



### Conclusion

The main goal of the project was to develop a custom scheduler for Kubernetes that optimized for a metric different from that of the default scheduler. The above experiments and the results show that the objective was clearly met. We developed a custom scheduler that successfully schedules pods on nodes with maximum memory available and tested it against the default one. While the default scheduler picks the node that *appears* to be least-loaded, based on pods’ CPU and Memory requests, our custom scheduler picks the node based on *actual* memory available.

Next steps could include exploring alternative monitoring tools, such as Weave Scope, Jaeger, The EFK Stack etc. to see if other relevant node metrics can be retrieved and optimized during scheduling. One could also consider using more than one metric to determine the best node. We could also extend the scheduler to track the memory usage on the nodes over time, since it can vary depending on how often the workloads are using more memory. Currently, we only consider the memory usage at the time a pod is scheduled. It could be more useful to schedule a pod on the node that had the lowest average memory usage. Our scheduler does not currently apply the priority functions that the default scheduler does. For instance, it will not try to prioritize spreading replicas of the same app across different nodes. In future we could extend the scheduler to use priority functions.

Overall, this was an interesting and challenging project. It involved a steep learning curve, given that none of us had any previous experience with Kubernetes or were even aware of the existence of the tools on our tech stack. The first couple weeks involved wading through multiple online resources / tutorials to assimilate essential Kubernetes knowledge, from basic terminology to simple kubectl (Kubernetes’ command line interface) commands. Many of the bottlenecks we hit, involved installation issues for the various tools, such as incompatibility with OS type or version.

We first had to find a way to create multi-node clusters on our local machines to be able to test scheduling. The default Minikube tool allows only a single-node cluster to be run locally. The most obvious way was to use VMs as nodes. The group members with Mac laptops ended up using Mirantis’ Kubeadm-dind-cluster (KDC) tool, that uses the Kubernetes kubeadm app and DIND (Docker-in-docker), to launch a cluster made of Docker containers rather than VMs. “This allows for faster restarts of the cluster” [11]. The group member using Windows used Kubernetes Engine on the Google Cloud Console to create a cluster, since only certain versions of Windows 10 are supported for Docker Desktop [14].

The hardest part, perhaps, was finding a suitable means to scrape and retrieve node metric data; we explored and discarded several tools (such as Sysdig) due to installation/pricing issues, before finally settling on the open source Prometheus tool. Prometheus alone, however, was not enough, since it does not inherently provide node metrics. We had to deploy a special exporter (node-exporter) to expose metric data for a node, for Prometheus to scrape. The next step was finding a way to run an instance of node-exporter on every node so that we could get data for all nodes; luckily, this part was fairly straightforward with the help of a Kubernetes *DaemonSet*.

Another major challenge was simulating a test setup to compare the two schedulers’ behaviors. First, some ‘setup’ pods had to be deployed to create a suitable initial state, in order to demonstrate the difference between the test pod being scheduled by the default vs custom scheduler. Many of our initial designs didn’t work because they were too complex, or involved working against the default scheduler, or required the use of our custom scheduler itself for the setup (an arguably iffy approach). Ultimately, we were able to come up with a simple, yet effective, test setup as described in the previous sections.

The actual coding constituted only a small part of the team’s time and effort spent on the project, and was probably the easiest step of the whole process. The most time and effort went towards hunting for appropriate tools, building up our tech stack and getting everything to work together. Our biggest takeaways have been our newfound understanding of concepts related to Docker, container scheduling and Kubernetes, experience with system monitoring tools, and a greater endurance for dealing with installation issues.

**Code:** <https://github.com/meeramurali/Custom-Kubernetes-Scheduler>

**Presentation Slides:** <https://docs.google.com/presentation/d/1bWn2stSNfZlXqeDUoB2YjAz91Hwf1dNvc01ofYK3aks/edit?usp=sharing>

### References

### [1] Kubernetes tutorial. Url: <https://www.tutorialspoint.com/kubernetes/index.htm>

[2] Kubernetes 101: Pods, Nodes, Containers, and Clusters. Url: <https://medium.com/google-cloud/kubernetes-101-pods-nodes-containers-and-clusters-c1509e409e16>

[3] Kubernetes concepts. Url: <https://kubernetes.io/docs/concepts/>

[4] Sysbench workload. Url: <https://wiki.gentoo.org/wiki/Sysbench#Using_the_memory_workload>

[5] Prometheus. Url: <https://prometheus.io/>

[6] Node Exporter. Url: <https://prometheus.io/docs/guides/node-exporter/>

[7] A Deep Dive into Kubernetes Metrics. Url: <https://blog.freshtracks.io/a-deep-dive-into-kubernetes-metrics-part-2-c869581e9f29>

[8] The Kubernetes Authors. *Managing Compute Resources for Containers*. url: <https://kubernetes.io/docs/concepts/configuration/manage-compute-resources-container/> (accessed: 06.10.2019)

[9] Eduar Tua. *Scheduler Algorithm in Kubernetes*. Url: <https://github.com/kubernetes/community/blob/master/contributors/devel/sig-scheduling/scheduler_algorithm.md> (accessed: 06.10.2019)

[10] Eduar Tua. *The Kubernetes Scheduler*. Url: <https://github.com/kubernetes/community/blob/master/contributors/devel/sig-scheduling/scheduler.md>. (accessed: 06.10.2019)

[11] 4 cool Kubernetes tools for mastering clusters. Url: <https://www.infoworld.com/article/3196250/4-cool-kubernetes-tools-for-mastering-clusters.html>

[12] GopherCon 2016: Kelsey Hightower - Building a custom Kubernetes scheduler. Url: <https://www.youtube.com/watch?v=IYcL0Un1io0>

[13] Hightower Toy Scheduler. Url: <https://github.com/kelseyhightower/scheduler>

[14] Google Kubernetes Engine. Url: https://cloud.google.com/kubernetes-engine/