Herodotus: Increasing Transparency and Observability of the Kubernetes Scheduler

Darwin Do
Yale University

Abstract

The Kubernetes scheduler is a bit of a black box. When a pod is created, it needs to be placed on a node for execution. The kube-scheduler component of the Kubernetes platform chooses the host node based on a filtering and ranking algorithm that takes into account many different attributes of the current cluster state [5]. This decision-making process is almost completely blind to the end-user. From an operator's perspective, the kube-scheduler simply chooses a node at will without telling the user *why* it chose that node. The Herodotus Scheduler is a custom Kubernetes scheduler that attempts to address this issue of transparency. It functions identically to the default Kubernetes scheduler in terms of scheduling logic, with the exception that it records data regarding its scheduling decision-making process. This data is then exposed outside the cluster in a readable form for both humans and machines.

1 Introduction

1.1 Kubernetes Scheduler

When the Kubernetes scheduler detects a pod that is not bound to any node, the scheduler begins a scheduling cycle for that pod to find a feasible node to place it on. The scheduling cycle consists of two main operations: filtering and scoring.

The filtering process finds a subset of nodes that can feasibly host the pod. The question of feasibility is addressed through a series of *filtering plugins* which are run across each node being tested. If all filtering plugins pass, the node is marked as "feasible" and continues on to the scoring step.

During the scoring step, a set of *scoring plugins* assign each node a score according to metrics such as resource balance, current load, regional spread, etc. Once all of the scores are calculated, they are normalized by their configured weights and summed together. The node with the highest score is selected and the scheduler goes on to bind the pod to that node [5].

The default Kubernetes scheduler contains a set of filtering and scoring plugins that are enabled by default. The pluggable nature of the scheduling framework also allows anyone to create filtering and scoring plugins of their own [6]. This was a major design consideration when developing Herodotus. Herotodus needed to have fine-grain detail to explain the logic of the scheduler on a per-plugin detail, but it also needed to take into account any third-party plugins that may be installed on other clusters.

1.2 Herodotus

The Herodotus system is composed of three components: the **herodotus-scheduler** scheduling binary, the **herodotus-endpoint** HTTP server, and a kubectl extension that allows users to easily interface with the Herodotus metrics.

The **herodotus-scheduler** is the replacement scheduling binary for the Kubernetes control plane. Users may choose to either replace the default **kube-scheduler** with **herodotus-scheduler**, or to run **herodotus-schduler** alongside the default. Should a user choose to do the latter, they will have to explicitly specify that a pod runs with the **herodotus-scheduler** in order for scheduling metrics to be recorded.

The **herdotus-endpoint** server is a simple Python HTTP server that runs on a pod in the kube-system namespace alongside **herodotus-schduler**. This server responds to HTTP queries regarding Herodotus metrics by fetching the data from **herodotus-scheduler** and returning it as a JSON object.

The kubectl plugin is an extension to the existing kubectl command line interface that will package HTTP requests to **herodotus-endpoint** and summarize the returned JSON data in a human-readable form.

2 Implementation

2.1 Collecting the Data

The **herodotus-scheduler** is a modified version of the default **kube-scheduler** built off of the Kubernetes source code. Most Kubernetes system components spawn a HTTP server and deliver Prometheus-style statistics on the /metrics endpoint. The **kube-scheduler** component is no exception. The **herodotus-scheduler** takes advantage of this already existing infrastructure to collect and export the decision-making metrics. From the Kubernetes source code, new metrics were registered at

```
    pkg/scheduler/metrics/metrics.go
```

The following custom metrics were added to **herodotus-scheduler**:

- **NodeNormalizedScoreTotal**: The total normalized score a node has accrued
- **NodeNormalizedScore**: Normalized score that a specified scoring filter gave to a pod for a node
- **NodeScoreAttempts**: Total amount of times a node has been scored
- **NodeFilterStatus**: The latest filter plugin status for a pod on a node
- **NodeFilterPasses**: The number of times a filter has passed on a node
- **NodeFilterAttempts**: The total number of times a filter has run on a node
- **NodeScoreByPluginTotal**: The total score a scoring plugin has accumulated on a node
- **NodeEligibleNum**: The number of times a node has passed all filtering plugins in a cycle
- **NodeEligibilityCheckNum**: The number of times a node has been checked for eligibility (i.e. the number of full scheduling cycles a node has gone through)

The following files were then modified to update each of the new metrics at the appropriate time:

```
pkg/scheduler/metrics/schedule_one.go
pkg/scheduler/metrics/framework/types.go
pkg/scheduler/metrics/framework/runtime/framework.go
```

For transparency and ease of grading, all custom code specific to Herodotus are marked by comments.

Kubernetes also allows scheduling plugins to pass along stateful data to each other throughout the scheduling pipeline [8]. A potential alternative to the current design is to modify each filter/scoring plugin manually to include the specific details that led up to the filtering decision or score. For example, I could have modified the **NodeResourcesBalancedAllocation** scoring plugin to include details about the exact resource usage of the current system that led to the score determined by **NodeResourcesBalancedAllocation**. However, this would require modifying every single scheduling plugin to pass along Herodotus state data. Any third-party plugins would need to be modified to be Herodotus-compliant as well. This is infeasible and too time-consuming, so I opted to go for a higher-level approach of purely summarizing plugin output in the **schedule_one.go** and **framework.go** files. This way, any third-party plugins registered through the pluggable scheduling framework will still work with Herodotus.

2.2 Retrieving the Data

To retrieve the metrics collected by **herodotus-scheduler**, the **herodotus-endpoint** was created. This endpoint is a Python HTTP server that runs on a pod in the same namespace as the scheduler. Two Kubernetes services are used to expose the scheduler to the endpoint, and the endpoint to the user. When a valid HTTP query is sent to the endpoint, the endpoint performs a GET HTTP request to the /metrics resource of **herodotus-scheduler**. The endpoint server then uses a series of regex filters to find the desired metrics, compiles them into a compact JSON object, and returns the JSON object to the client.

The endpoint is exposed to the client directly through a NodePort Kubernetes service. The kubectl herodotus plugin is an easy way to interface with the endpoint, allowing the user to query the endpoint API directly through the command line. The kubectl plugin automatically detects which port the NodePort service is exposed on and runs HTTP requests against that node. Both the endpoint and kubectl plugin have error checking so that incorrect user queries return helpful error messages and don't crash the services.

This design was chosen for its modularity, stability, and conformance to best practices. One alternative design would be to expose the /metrics resource on **herodotus-scheduler** directly to the user without the need for the endpoint. However, this didn't seem like a good idea as it broke the encapsulation of one of the core Kubernetes system components. It also exposes a whole slew of other non-Herodotus metrics to the world which may not be desirable. Another design choice could be to create a sidecar command line interface process that fetches and outputs the summarized metrics itself. This sidecar process would function similarly to **herodotus-endpoint**, except that there would be no HTTP server. Users would call commands directly into the container hosting the process by running kubectl exec commands. One benefit of this method would be that there would be no need to manage a second **herodotus-endpoint** service. Another benefit is that it would more easily allow the sidecar process to use non-standard-library Python tools such as the *requests* package for easy HTTP requests and the *termcolor* package for easy ANSI-compliant color printing to the terminal. However, the concept of hosting a pod just to run "kubectl exec" against conflicts with the philosophy of having a container exist for a single purpose and ending when that job has completed.

Creating a HTTP server on the cluster to serve requests into /metrics is a good solution as it does not over-expose the metrics on **herodotus-scheduler**. An HTTP server is a standard job to run on a container. As long as the server can and should serve requests, the container will be alive. Since this middle-man technique also exposes the information as a JSON object, this architecture is extendable and one could easily make a separate program to interact with the **herodotus-endpoint** API and display or aggregate the data in other means.

3 Usage

The exact syntax of the kubectl herodotus CLI can be found in the README of the codebase and the help page of the plugin.

3.1 Querying Pods

The scheduling info for a specific pod can be queried with the following command:

```
kubectl herodotus pod [POD_NAME] -n [POD_NAMESPACE]
```

If the namespace is set to "default" if the flag is omitted. This command returns the status of all the filtering plugins run against each node in the cluster. If all the filters passed for a specific node, a one-line message is printed stating as such for brevity's sake. Otherwise, a comprehensive breakdown is displayed showing the filter plugin that failed, the ones that passed, and the ones that were skipped.

The scoring breakdown for each node is also displayed. This breakdown shows the total score that the scheduler gave each node and the score from each individual scoring plugin. If only one node is available after the filtering process, the scheduler automatically picks that node without running the scoring plugins.

The bottom section of the pod query output displays the node that the scheduler ended up choosing and the reason. Unless only one node was available, this reason will always simply be that the chosen node had the highest score. If there is a tie for the highest score, the scheduler will choose one of the highest-scoring nodes at random.

This command is meant to be used to see how a specific pod was placed on a specific node.

3.2 Querying Nodes

The scheduling info for a specific node can be queried with the following command:

```
kubectl herodotus node [NODE_NAME]
```

This command returns aggregated statistics for the selected node. The output shows the total number of scheduling cycles run against the node and the number of scheduling cycles where the node passed all filtering plugins. The output also shows a breakdown of all the score a node has accumulated and the individual scoring plugins that have contributed to that total. At the bottom, the output shows the breakdown for all the filter plugins that have run on this node and their corresponding pass rate.

This command is meant to be used as a way to look at general trends and to potentially identify "hot" or "cold" nodes that get frequently or infrequently respectively.

3.3 Edge Cases

While this system is very robust in capturing the statistics of scheduler decisions, there are some edge cases where pods may be placed on nodes without being detected by Herodotus.

The Kubernetes scheduler framework has a system called preemption where a lower-priority pod may be evicted to make space for an incoming higher-priority pod if there are no nodes available for the higher-priority pod [4]. If a pod is selected through preemption, this goes through a different scheduling cycle and the Herodotus logic will not pick up on it.

A pod can also manually specify which node it wants to be placed on through the *nodeName* field in the pod specification. If this field is set, Kubernetes will automatically place the pod on that node if it is available without touching the scheduler at all [7].

4 Related Work

The Herodotus scheduler builds off of a lot of pre-existing work relating to scheduling and metrics in the Kubernetes system.

The Kubernetes developers made the scheduler framework modular by default. Their intention was to allow for the use of other developers to create third-party scheduling plugins that could fit their own organization's needs. Many of these out-of-tree plugins can be found on a Kubernetes special-interest-group hosted Github repository [1]. As mentioned earlier, the Herodotus scheduler aims to mimic that level of modularity by gathering statistics at the plugin-output level rather than the plugin level itself.

Herodotus also builds off of existing work with metrics related to the Kubernetes ecosystem. The default **kube-scheduler** emits metrics on the /metrics endpoint of its self-hosted HTTP server. These metrics are in text form and use the Prometheus format. They also use Prometheus metric types such as counters, gauges, histograms, and more [2] [3]. Herodotus leverages this existing infrastructure to effectively communicate metrics outside of the scheduler pod. All of the Herodotus metrics use Prometheus counters and gauges with appropriate labels to separate metrics by pod/node/plugin.

The inspiration for this project comes from personal experience when I was an intern on a DevOps team for a small technology company in Boston. My team operated a Kubernetes cluster with dozens of nodes and hundreds of pods spinning up and down daily. My team had labeled a subset of nodes to prefer to run pods of a specific job type. However, they found that the Kubernetes scheduler often elected to place these specific pods into other nodes that did not have this label. As an intern with a lot of free time, I was tasked with investigating this phenomenon.

I found this task difficult as the scheduler logs left no trace of what decisions it was making. Even with setting the scheduler to have the maximum log verbosity, I could not figure out what was going on. It was only when I decided to re-compile the scheduler binary from scratch with many more manually placed log commands did I discover that the scheduler couldn't place the specific pods on the labeled nodes because the labeled nodes often had the maximum number of 110 pods already bound to it.

While this was a great and fun learning experience, I still wished it had been easier to see what exactly was going on with the scheduler's decision-making process. The Herodotus Scheduler is my solution to this problem.

References

[1] The Kubernetes Authors. Scheduler-Plugins. Cloud Native Computing Foundation. Retrieved from https://github.com/kubernetes-sigs/scheduler-plugins

[2] The Kubernetes Authors. Metrics For Kubernetes System Components. Cloud Native Computing Foundation. Retrieved from https://kubernetes.io/docs/concepts/cluster-administration/system-metrics/

[3] Prometheus Authors. Metric Types. The Linux Foundation. Retrieved from https://prometheus.io/docs/concepts/metric_types

[4] The Kubernetes Authors. Pod Priority and Preemption. Cloud Native Computing Foundation. Retrieved from https://kubernetes.io/docs/concepts/scheduling-eviction/pod-priority-preemption/#preemption

[5] The Kubernetes Authors. Pod Priority and Preemption. Cloud Native Computing Foundation. Retrieved from https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/

[6] The Kubernetes Authors. Pod Priority and Preemption. Cloud Native Computing Foundation. Retrieved from https://kubernetes.io/docs/reference/scheduling/config/

[7] The Kubernetes Authors. Pod Priority and Preemption. Cloud Native Computing Foundation. Retrieved from https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/

[8] The Kubernetes Authors. Kubernetes Source Code (Release v1.26.0) [Source code]. Cloud Native Computing Foundation.