Predictive Autoscaling of Pods in the Kubernetes Container Cluster Manager

Matthew McNaughton
Advisor Jeannie Albrecht
Computer Science Thesis, Williams College

October 22, 2015

1 Background and Motivation

Society is becoming increasingly dependent on computation. Every facet of human life, be it finance, healthcare, education, social interactions to name just a few, now involves computers. [9, pg. 4] Naturally, singular computing workstations cannot provide the computational power necessary to execute the demanded tasks at such a massive scale. [11, pg. 2] However, cluster computing, in which commodity level PCs linked by a high-speed LAN offer a singular mass of resources, can provide the requisite performance. [11, pg. 17] A cluster is used to run applications, and a cluster manager is responsible for admitting, scheduling, starting, restarting, and monitoring the specified applications on the cluster. [12, pg. 1]. While cluster computing was largely researched and implemented at large private corporations in the past, it is increasingly becoming available to the general public. Kubernetes, an open source cluster manager from Google, leads this effort. [8] Interest in, and the importance of, cluster computing will only increase as it becomes more accessible to common programmer and more vital to processing the ever growing mass of computing tasks.

Kubernetes manages all aspects of the cluster through admitting, running, and restarting applications, monitoring and displaying application health, and maintaining the underlying machines composing the cluster. Understanding the Kubernetes method of cluster management requires understanding terms specific to this cluster manager. Most importantly, Kubernetes defines the pod as the smallest deployable unit of computing. A pod contains containers and data volumes: a container is a single application coupled with the complete file system necessary for execution, while a data volume is a persistent directory intended to preserve data beyond the variable lifetime of the container. [2] Containers, and the applications within them, are intended to be started, stopped, deleted and replicated at will, without risking the loss of important data, hence the need for persistent data volumes. A pod can comprise of a collection of containers and data volumes designed to perform a singular job, although typically a pod includes a single container, and is responsible for performing the job the application within the container dictates. Pods can be used to deploy web servers, databases, data-processing frameworks, and more. [5] Additionally, Kubernetes defines the concept of a replication controller, which simply ensures a given number of specified pods are running at all times. [6] Replication controllers are used in conjunction with services, which provide a long-lasting, logical abstraction for a collection of identical temporary pods. Services provide load balancing from any incoming request to the pods the service abstracts. [7] For example, a service would abstract multiple pods all running the Apache web server by passing incoming requests to a different singular pod each time. Understanding the concepts of pods, replication controllers, and services is necessary for any work with Kubernetes.

Services have various performance requirements and face ever changing external factors. For example, a service abstracting multiple pods running a web server application may be asked to process a highly variable number of requests. As requests spike, an ever increasing amount of computation is spread across the pods. If the replication controller defines a constant number of replication pods and the load to the service increases, each pod will be asked to a use more resources. If the constant replication value is not high enough, one or more pods may attempt to compute outside of its capacities with respect to CPU or memory. In contrast, if the constant replication value is too high, and the load is balanced such that no pod operates anywhere close to capacity, costly computing resources are wasted. In many use cases, one or more pods operating below or beyond their capacity is detrimental to the health and efficiency of the service and cluster.

Kubernetes solves the dual concerns mentioned above through varying the amount of pod replication, a common method of scaling distributed systems. [11, pg. 15] More specifically, Kubernetes implements an optional behavior entitled horizontal pod autoscaling, which works as follows. Consider a service S balancing load to a set of identical pods $P = \{p_1 \cdots p_n\}$. For any pod $p_i \in P$, we define a target value for resource consumption, signified as $target(p_i)$. An example target resource consumption value could be 80% CPU usage or 100 mb of RAM. At a specified interval, the autoscaler checks the resource usage across all pods, defined as $curr(p_i)$ for all $p_i \in P$, and adjusts the number of replicas so that average resource usage conforms to target resource usage. Ultimately, given a current set of pods $P_c = \{p_1 \cdots p_n\}$, the target set of pods is $P_t = \{p_1 \cdots p_m\}$, where $m = \sum_{i=0}^{n} curr(p_i)/target(p_i)$. For example, if there currently exists $P = \{p_1\}$ such that $curr(p_1) = 100\%$ CPU usage, while $target(p_i) = 50\%$ CPU usage, the autoscaler would replicate p_1. With the existence of this replica p_2, such that $P = \{p_1, p_2\}$, we are now ensured $curr(p_1) = curr(p_2) = target(p_i)$. The exact same process works for scaling down if the current CPU usage of each pod is too lower compared to the target resource usage. [4]

While horizontal pod autoscaling is useful in increasing Kubernetes' ability to handle varying loads, autoscaling could be more responsive. Specifically, the current method of autoscaling fails to consider the amount of time necessary to create a pod, which we will call $create(p_i)$. [1] $create(p_i)$ is a function of the complexity and build time for the containers within the pod p_i. For example, the create time for pod consists of a container running a simple C program may be only a couple of seconds, while the create time for a pod comprised of multiple containers each utilizing a complex Java library may be minutes or in extreme cases hours. To illustrate the necessity of considering pod creation time, consider the following example. To begin, consider again a service S balancing load to a set of identical pods $P = \{p_1\}$, such that $create(p_i) = 3000s$ and $target(p_i) = 50\%$ CPU usage. Imagine at $t = 0s, curr(p_1) = 100\%$ CPU usage. Naturally p_1 operating above target resource consumption suggests replicating p_1 so that $P = \{p_1, p_2\}$ and $curr(p_1) = curr(p_2) = target(p_i)$. Yet because $create(p_i) = 3000s$, it is not until $t = 3000s$ that the replication is complete, load can be balanced across

$P = \{p_1, p_2\}$, and resource consumption returns to the target. In certain contexts, particularly those in which operating outside of target resource consumption is damaging or the pod's creation time is non-trivial, this delay is extremely detrimental. Clearly, work can still be done to increase Kubernetes' scaling responsiveness.

2 Research Question

This thesis will seek to address deficiencies previewed in the motivation section. Specifically, we will answer the following question: How can we decrease the amount of time pods spend operating outside of target resource consumption? Success in answering this question will include the scaling responsiveness of Kubernetes and increase its ability to respond to variable external factors.

3 Hypothesis

We hypothesize it is possible to decrease the amount of time pods spend operating outside of target resource consumption through predictive autoscaling. Predictive autoscaling attempts to scale based on the pods' future, not present, resource consumption, so that a pod required in the future can begin the creation process before any pod operates outside the target value, and be functional as soon as it is needed. More formally, consider again the example given at the end of the previous section. This time, imagine that instead of beginning the replication process at $t = 0s$, when $curr(p_1) = 100\%$ CPU usage, we instead predicted at $t = -3000s$ that at $t = 0s, curr(p_1) = 100\%$ CPU usage, and we began to create p_2. After the $3000s$ necessary for the creation of p_2, both pods would be available, and $curr(p_1) = curr(p_2) = target(p_i)$. As opposed to if we waited until $t = 0s$ to begin creating p_2, we do not have to spend any time operating outside of target resource consumption. Predictive scaling increases the responsiveness of autoscaling by decreasing the impact of pod creation times, and thereby minimizing the amount of time any pod must function beyond target resource consumption.

4 Implementation and Methodology

Kubernetes is a rare and interesting project in that it is both completely open source and in production use at Google. [3] Thus, we will be implementing our work with predictive autoscaling on the actual Kubernetes code, with the eventual goal of our work becoming part of the production Kubernetes distribution.

Considering our goal of merging this code into a system as heavy-utilized as Kubernetes, it is important to show tangible successes or failures through evaluation. Fortunately, a fairly defined methodology exists for evaluating cluster managers like Kubernetes. [10, pg. 355] The first aspect of evaluation utilizes a lightweight simulator based on the statistical structure of typical input to Kubernetes. For example, testing the ability of Kubernetes implementing predictive autoscaling to respond to a spike in web traffic, would require simulating the highly variable incoming web requests through statistical modelling. The second evaluation replays historic data, in order to make exact conclusions about how Kubernetes with predictive autoscaling would have operated in a specific situation. This analysis is typically more restrictive, because it can be difficult, and time-consuming, to obtain and replay exact data.

Evaluation will involve tracking the internal metrics Kubernetes already records, including the amount of time a pod is running, and the specific computational metrics for the pod (ie, CPU, memory...). Work will be necessary to decide which metrics are best suited to measuring scaling responsiveness.

5 Current State

Preliminary work with Kubernetes has been extremely encouraging. Most importantly, the lead developer of Kubernetes, Brendan Burns '98, is a Williams graduate who has been very generous with his time and expertise as we have discussed potential areas of exploration. I am hopeful that I will be able to continue working with Brendan going forward.

Naturally, Kubernetes is a large open source project, and as such there are certain standards and customs which must be followed. Based on past open source work and Kubernetes extensive documentation, I have already made small contributions to Kubernetes, and am hopeful about the possibility of making larger contributions, including the predictive scaling examined in this thesis.

References

[1] Conversation with brendan burns, lead developer of kubernetes.

[2] Docker website. https://www.docker.com/.

[3] Google container engine. https://cloud.google.com/container-engine/docs/.

[4] Kubernetes horizontal pod autoscaler proposal. https://github.com/kubernetes/kubernetes/blob/master/docs/proposals/horizontal-pod-autoscaler.md.

[5] Kubernetes pods documentation. http://kubernetes.io/v1.0/docs/user-guide/pods.html.

[6] Kubernetes replication controllers. http://kubernetes.io/v1.0/docs/user-guide/replication-controller.html.

[7] Kubernetes services. http://kubernetes.io/v1.0/docs/user-guide/services.html.

[8] Kubernetes website. http://kubernetes.io.

[9] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed Systems: Concepts and Design*. Addison-Wesley Publishing Company, USA, 5th edition, 2011.

[10] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *SIGOPS European Conference on Computer Systems (EuroSys)*, pages 351–364, Prague, Czech Republic, 2013.

[11] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms (2Nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[12] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.