

Predictive Pod Autoscaling in the Kubernetes Container Cluster Manager

by

Matt McNaughton

Professor Jeannie Albrecht, Advisor

A thesis submitted in partial fulfillment
of the requirements for the
Degree of Bachelor of Arts with Honors
in Computer Science

Williams College
Williamstown, Massachusetts

November 12, 2015

DRAFT

Contents

1	Introduction	5
1.1	Goals	6
1.2	Contributions	7
1.3	Contents	7
2	Background	8
2.1	Resource Intensive Computing Paradigms	8
2.2	Cluster Management Paradigms	9
2.2.1	Borg	11
2.2.2	Omega	11
2.2.3	Mesos	11
2.2.4	YARN	12
2.2.5	Kubernetes	12
2.3	Auto-scaling Paradigms	13
2.3.1	Threshold-based Rule Policies	13
2.3.2	Time-series Analysis	13
2.3.3	Control-Theory	13
2.4	Summary	13
3	Appendix	14
3.1	Technologies Underlying Kubernetes	14
3.1.1	Containerization	14
3.2	Trends Motivating Kubernetes	14
3.2.1	Microservices	14

Abstract

Acknowledgments

Chapter 1

Introduction

Over the past few decades, an explosion in the need for computing resources, and the existence of cheap, interconnected computers, has driven a significant increase in the feasibility and benefits of distributed systems. [13, pg. 1]

First, we consider the origin of distributed systems as a field of computer science. Before the availability of cheap, powerful microprocessors and reliable, efficient local-area networks (LANs), computational tasks could only be performed on a singular computer. [13, pg. 1] If a task was too computationally expensive for a commodity PC, the only solution was to run it on a larger, more powerful supercomputer. However, as cheap microprocessors increased computers' availability, and LANs fostered quick inter-computer communication, a new model of performing resource intensive computation, distributed systems, arose. In the distributed systems model, a collection of individual computers function as a single computer to solve a given computational task. [13, pg. 2]

Second, we consider the ever-growing interest in unlocking and implementing the benefits of distributed systems. A number of forces drove, and continue to drive, increased interest in distributed systems over the past decade. The first, and most obvious, factor is the Internet. As more people connected to the Internet, through computers, mobile phones, and tablets, an increasing number of human interactions became computerized. Consumption, communication, research, and more all became possible on the Internet. Naturally, large amounts of computing resources were needed to store the data, and perform the computational tasks, related to these interactions. Closely coupled with this trend is the rise of "Big Data". In 2013, the digital universe contained 4.4 zettabytes of data.¹ [8] Naturally, without multiple computers working together it would be impossible to store and process this incredible volume of data. Today, it is nearly impossible to do anything in modern society without interacting with a distributed system and creating new digital data. Driving a car, trading a stock, visiting a doctor, checking an email, and even playing a simple video game, are all activities that distributed systems facilitate and improve. [9, pg. 4] As life becomes more computerized, and as the volume of data humans generate and hope to process grows, distributed systems will only increase in importance. Furthermore, research into distributed systems makes it possible to continue to unlock, and make available to the general public, the incredible power of networked,

¹A zettabyte equals 10^{21} bytes, which equals 1 billion terabytes.

cooperating computers. As the distributed systems supplying massive computational power become more accessible, both because of decreased cost and increased ease of use and reliability, we can computationally address an ever increasing number of challenging, important problems.

There are a number of different models for computing tasks requiring high levels of computing resources, including supercomputing, cluster computing, and grid computing. In this thesis, we focus on cluster computing. Cluster computing groups together similar commodity PCs on the same LAN to offer a singular mass of computing resources. Specifically, we focus on the cluster manager, an integral component of cluster computing. Cluster managers are responsible for abstracting all of the management details of the distinct nodes in the cluster, and instead presenting a single mass of computing resources on which the user can run jobs or applications. In other words, a cluster manager “admits, schedules, starts, restarts, and monitors the full range of applications” on the cluster. [16, pg. 1] There are a variety of different cluster managers, the most important of which will be discussed in the background chapter, each pursuing different objectives. This thesis will ultimately focus on Kubernetes, an open-source cluster manager from Google. [5]

Cluster managers seek to accomplish a number of different goals, and as a result, multiple metrics indicate success. For example, Microsoft’s Autopilot is predominantly concerned with application uptime, and thus success is measured with respect to reliability and downtime. [11, pg. 1] Alternatively, a number of cluster managers measure themselves based on efficient resource utilization (ERU). [16, pg. 7] Essentially, efficient resource utilization relates to the percent of cluster resources which are actually being used. One such measurement of this goal, cluster compaction, examines how many computers could be removed from the cluster, while still comfortably running the cluster’s current application load. [15, pg. 5] This metric is particularly important, because the more efficient the cluster management is at utilizing resources, the less clusters cost, and the more accessible cluster computing becomes to the general public. A final important cluster management metric is quality of service (QOS). Quality of service measures the ability of an application to function at a specified performance level, despite ever-changing external factors. Again, this metric is particularly important because increasing the robustness of applications run on cluster managers means these applications can be trusted with increasingly important tasks. Cluster managers predominantly differ with respect to which metrics they optimize for, and the process by which this optimization occurs.

1.1 Goals

This thesis is most concerned with maximizing the efficient resource utilization (ERU) and quality of service (QOS) metrics with respect to the Kubernetes cluster manager. As such, this thesis pursues three goals:

1. Given an application running on a Kubernetes cluster, we seek to determine a method which ensures quality of service stays consistently high regardless of external factors. While it is difficult to make guarantees regarding quality of service, because application performance is dependent on a number of uncontrollable, varying external factors, it is possible to ensure each

application has, and is utilizing, the resources it needs to function properly. Given the cluster manager grants the application the resources it needs to function given the current external load, the cluster manager has done all it can to ensure a high quality of service.

2. A simplistic solution to the first goal of ensuring a high application quality of service is to just give each application many more resources than it requests. Yet, this overallocation is inefficient and costly. Thus, our methods for ensuring a high quality of service must also ensure the maintenance, or improvement, of the efficient resource utilization metric. Thus, we add an additional goal: given a certain number of applications running on a Kubernetes cluster, we seek to determine a method which ensures the cluster is as small as possible, while still comfortably supporting the application's current, and future, resource needs.
3. Given Kubernetes is an open-source project, we seek to implement, test, and evaluate a proposed enactment of the previous two goals. Thus, the methods we pursue will in part be dictated by the current structure and implementation of Kubernetes. Tests will be conducted using the Google Compute Engine [2] on both simulated and real Kubernetes user data. The eventual goal is for this thesis' improvements to be merged into the production version of Kubernetes used to run 1000s of applications at Google everyday.

1.2 Contributions

This thesis presents our given contributions to Kubernetes. Kubernetes seeks to ensure high application quality of service and efficient resource utilization, and our contributions look to further its ability to accomplish these goals. As such, we present not only new methodology, but also new, working implementations with the accompanying evaluation. We demonstrate the effectiveness of our modifications in comparison to the non-modified Kubernetes using both simulated and real-world datasets. Finally, we discuss the experiences of making these modifications to Kubernetes, as well as avenues for future improvements with respect to Kubernetes and cluster managers in general.

1.3 Contents

@TODO - This section can not be written until the thesis is completed.

Chapter 2

Background

2.1 Resource Intensive Computing Paradigms

As was briefly mentioned in the introduction, a number of different paradigms exist for undertaking computing tasks too resource intensive for a single computer. They are discussed in detail below:

1. Supercomputing: The supercomputing model responds to increased demands for computing resources by increasing the technical specifications of the computer far beyond the range of the traditional commodity PC. While supercomputers are able to avoid the majority of the complications resulting from the introduction of networks, most prominently reliability and security, there are naturally limits on the power of supercomputers. Importantly, constructing supercomputers is extremely expensive, and thus their computing power is not available to the general public. Furthermore, it is difficult to scale a supercomputer should the need arise. Finally, supercomputers offer a single point of failure, meaning they are not particularly robust to error. These limitations have decreased the usage of supercomputers to provide the mass of computing power needed in the “Big Data” era.
2. Cluster Computing: Cluster computing is defined as utilizing “a collection of similar workstations of PCs, loosely connected by means of a high-speed local-area network [where] each node runs the same operating system.” [13, pg. 17-18] Cluster computing can provide a mass of computing power similar to that contained in a supercomputer. Cluster computing also offers many advantages over the single supercomputer. First, and perhaps most importantly, they are much more cost-efficient, and thus much more accessible. Second, clusters are easy to scale by simply adding new commodity PCs as nodes. Finally, cluster computing is much more fault tolerant, as a single failing commodity computer will simply be removed from the cluster. Cluster computing is used in the implementation of what is colloquially referred to as *Cloud computing*, in which large amounts of computing resources are offered on a per-usage basis. [9, pg. 13] Cloud computing, as implemented by Amazon Web Services, [1] Microsoft Azure, [6] and Google Compute Engine, [2] continue to revolutionize the development and

deployment of computing applications, as developers gain access to cheap, easily accessible, and quickly scalable computing power.

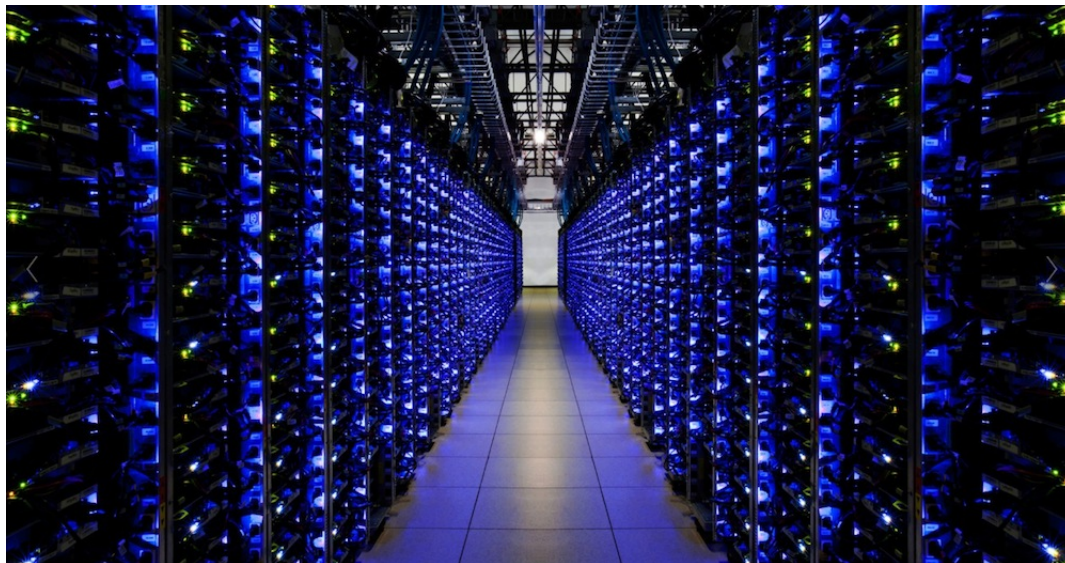


Figure 2.1: A Google Computing Cluster. [7]

3. Grid Computing: Grid computing is similar to concept in cluster computing, except it foregoes the requirement that all computers within the grid be relatively homogeneous. As such, the grid computing model accounts for a large degree of heterogeneity with respect to network membership, operating system, hardware, and more. [13, pg. 18] While grid computing systems lack of homogeneity requirements increase flexibility, the resulting heterogeneity introduces significant complexity.

Ultimately, because of simplicity, cost, and scalability, cluster computing is the most prominent resource intensive computing paradigm. Thus, cluster computing, and the accompanying cluster manager, is the focus of this thesis.

2.2 Cluster Management Paradigms

As was briefly mentioned in the introduction, cluster managers are responsible for admitting, scheduling, running, maintaining, and monitoring all applications and jobs a user wishes to run on the cluster. Naturally, cluster managers are extremely diverse, both in the types of applications and jobs they are most suited to running, and the method in which they seek execute their duties. At the most basic level, there are two types of workload that may be submitted to a cluster manager: production and batch. Production tasks are long-running with strict performance requirements and penalties to downtime. Batch tasks are more flexible in their ability to handle short-term performance variance. In the context of a large company like Google, a production task would be serving

a large website like Gmail, which must be continuously accessible with low-latency and little downtime. A batch task would be analysing advertising analytics data with MapReduce, which can fail or slow without significant external costs. [16, pg. 1] The type of tasks a cluster management system predominantly seeks to run dictate the cluster manager’s implementation details.

One important decision in the implementation of a cluster manager is the manner by which the cluster manager schedules jobs.¹ There exist three different methods of scheduling: monolithic, two-level, and shared state. With monolithic scheduling, a single algorithm is responsible for taking the resource requests of all jobs and assigning them to the proper machine. With two-level scheduling, the cluster manager simply offers resources, which can then be accepted or rejected by the distributed computing frameworks.² Finally, with shared state scheduling, multiple different algorithms concurrently work to schedule jobs on the cluster. [12, pg. 1] Naturally, all of these methods have positives and negatives. While monolithic scheduling is simple to initially, a single-threaded monolithic scheduler does not allow nuanced processing of jobs. Attempts to add this nuance can create an incredibly complicated algorithm that is difficult to extend. [12, pg. 353-354] While two-level scheduling is lightweight, simple, and offers advantages with respect to data locality, it is not effective for long-running, production jobs. Finally, while shared state scheduling removes the scheduler as both a computational and complexity bottleneck, yet must take steps to guarantee global properties of the cluster. [12, pg. 363] The ultimate determined method of assigning a job resources effects the type of applications and distributed computing frameworks runnable on the cluster manager and the efficiency with which these applications and frameworks run.

A final distinction is licensing and availability of the cluster manager’s code. Because entities need cluster managers only to process and store massive amounts of data and human-computer interaction, mainly large corporations develop and utilize cluster managers. Often these cluster managers are kept within the confines of the corporation, or only explained by a brief paper or conference talk, with little source code available. In more unique cases, the company will open-source the source code, allowing anyone to view, modify, and run the cluster manager. Such open-sourcing presents a unique opportunity for researchers wishing to experiment with cluster managers, yet without the resources to create their own from scratch. In rarer instances, a fully-developed cluster manager will originate from academic research. In unique scenarios, a large corporation will use this cluster manager and the full code will be open-sourced. The availability of source code directly impacts the feasibility of pursuing experiments with an already existing cluster management system.

Naturally, cluster managers can vary in multiple additional ways. However, the previous three differences recognize the most important distinctions in this thesis’ context. Given this understanding, we can now begin to examine specific cluster management implementations and defend our choice of Kubernetes as the cluster manager on which we will ask and answer our research question.

¹Scheduling jobs on machines simply equates to assigning jobs to resources on a machine.

²Distributed computing frameworks are frameworks built to function over multiple machines: Apache Hadoop, Apache Spark...

Table 2.1: Overview of Cluster Management Paradigms.

	Job Type	Scheduling Model	Open Source
Borg	Both	Monolithic	No
Omega	Both	Shared state	No
Mesos	Batch	Two-level	Yes
YARN	Batch	Monolithic	Yes
Kubernetes	Production	Monolithic	Yes

2.2.1 Borg

While just recently described to the public in a 2015 paper, the *Borg* cluster manager from Google has been in use for over a decade. [16, pg. 14] Borg incorporates many different objectives, seeking to abstract resource management and failure handling, maintain high availability, and efficiently utilize resources on the cluster. [16, pg. 1] It is responsible for managing the hundreds of thousands of batch and production jobs run everyday at Google. Additionally, Borg utilizes a monolithic scheduler, as jobs specify the resources they need, and a single Borg scheduler decides whether to admit and schedule said jobs. Finally, Borg is not open-source. In fact, it was not even publicly announced or described until 2015, despite running at Google for over a decade. However, Kubernetes, Google’s open-source cluster manager, incorporates much of the work done to implement and improve Borg.

2.2.2 Omega

Also originating at Google, *Omega* is a cluster manager seen as the current-day extension of Borg. While retaining the mission and goals of Borg, Omega differs in implementation. Specifically, it considers a new method of assigning jobs the resources they need on the cluster by implementing shared state, instead of monolithic, scheduling. As mentioned in the general overview, shared state scheduling allows multiple scheduling algorithms to work in parallel to assign tasks the resources they request. Research on Omega shows this parallel scheduling implementation both eases the complexity of adding new scheduling behaviors and offers competitive scaling performance. [12, pg. 358-359] Additionally, in Omega, all schedulers are aware of the entire state of the cluster, meaning that the resource allocation to a job can be varied after the job begins to execute. This flexibility offers significant performance improvements. [12, pg.352] Like Borg, Omega is not open-source. However, like Borg, much of the work done on Omega is incorporated into Kubernetes, Google’s open-source cluster manager.

2.2.3 Mesos

Lest we think all cluster managers originate at Google, we now examine Apache Mesos, a cluster manager originating at University of California, Berkeley, and currently at use at a number of prominent corporations including Twitter. Mesos is much more lightweight than either Borg or Omega, and simply seeks to offer resources to distributed computing frameworks (i.e. Apache Hadoop, Apache Spark. . .) which can decide to accept or reject these resources. It does not seek to monitor the health of jobs or provide user-interfaces for viewing the current state of a job, leaving

those tasks to the distributed computing framework. Additionally, Mesos predominantly focuses on quick-running, high-volume batch jobs, and is not particularly suited to long-running, high-availability production jobs. [12, pg. 358] In part, Mesos’ job scheduling implementation specifies the singularity of the job types Mesos efficiently processes. Mesos utilizes two-level scheduling, in which the cluster management system simply offers, not assigns, available resources to the distributed computing framework. Predominantly, this decision is made to ensure data locality. [10, pg. 1] Data locality is a measure of if the task has the necessary data on its machine, or if it must make a costly request across the network for the data. Mesos works to ensure data locality by allowing multiple frameworks to function on the same machines, and thus share data, and also for frameworks to dictate their own resources, such that they can work to ensure data locality. If running a large number of data processing tasks, with extremely high volumes of data, data locality can be essential to efficient cluster operation. Finally, Mesos is interesting in that it is entirely open-source, yet still in use at some of the largest tech companies.

2.2.4 YARN

We now briefly discuss Apache Hadoop YARN. YARN, an acronym for *Yet Another Resource Negotiator*, is a cluster manager initially built for use with Apache Hadoop. [14, pg. 1] However, it is now possible to use YARN with a variety of distributed computing frameworks. Unsurprisingly given YARN’s initial use case, YARN is predominantly used for running batch jobs.³ Like Mesos, YARN aims to support data-locality, again a predominant advantage for batch processing. [14, pg. 3] YARN schedules resources by allowing per distributed computing framework application masters to request resources from a single resource master. [14, pg. 5] As a single resource master is assigning all of these resources, YARN is a monolithic scheduler. Similar to Mesos, YARN is both entirely open-source and in use at major corporations like Yahoo. [14, pg. 9]

2.2.5 Kubernetes

Finally, we arrive at the cluster management that is the focus of this thesis: Kubernetes. Kubernetes also originates, although it is open source, meaning anyone is free to modify and deploy Kubernetes. Kubernetes is also the most recent of the cluster managers we consider in this background research section.⁴ The recent explosion in popularity of containerization⁵ heavily impacts the development and implementation of Kubernetes. Specifically, Kubernetes is seen as part of a new paradigm of developing applications through the use of microservices.⁶ Kubernetes predominantly focuses on effectively running service jobs. As previously mentioned, these jobs require high-availability and potentially varying amounts of resources. Additionally, Kubernetes currently utilizes a monolithic,

³At the time of the publication, YARN was just beginning to be used for production jobs; however, its main focus from creation has been batch processing. [14, pg. 11]

⁴Kubernetes became public in the summer of 2014.

⁵We discuss both the motivations and technology behind containerization in the Appendix. Briefly, containerization is packing everything an application needs to run into a single *container* and then running that container on any desired computer.

⁶Again, we will discuss microservices in greater detail in the Appendix. Essentially, microservices are the division of applications into small, easily scalable services which communicate with each other across the network.

fairly simple scheduler, although there are plans to grow the scheduler in the future. [4] Finally, Kubernetes is an open source project, yet is also used in production at Google, and available to the public through the Google Container Engine. [3]

We choose Kubernetes as the cluster manager on which to conduct our experiments for a number of reasons. First, Kubernetes focuses on service jobs. Services jobs have particularly stringent requirements for availability and are also the most likely to have varying resource needs. Both these conditions are closely linked with the previously stated goals of this thesis, and Kubernetes is the cluster manager that stands to benefit the most if we achieve our goals. Additionally, Kubernetes is the only cluster manager focusing on long-running services that is open source. Focusing on an open source cluster manager allows us to benefit from the previous work of others, as well as expand the potential benefits of any successful work.

2.3 Auto-scaling Paradigms

2.3.1 Threshold-based Rule Policies

2.3.2 Time-series Analysis

2.3.3 Control-Theory

Model Predictive Control

2.4 Summary

Chapter 3

Appendix

3.1 Technologies Underlying Kubernetes

3.1.1 Containerization

Virtualization

Containerization

Docker

3.2 Trends Motivating Kubernetes

3.2.1 Microservices

Bibliography

- [1] Amazon web services. <https://aws.amazon.com/>.
- [2] Google compute engine. <https://cloud.google.com/compute/>.
- [3] Google container engine. <https://cloud.google.com/container-engine/docs/>.
- [4] Kubernetes design overview. <http://kubernetes.io/v1.1/docs/design/README.html>.
- [5] Kubernetes website. <http://kubernetes.io>.
- [6] Microsoft azure. <https://azure.microsoft.com/en-us/>.
- [7] Google cluster. <https://talks.golang.org/2012/splash/datacenter.jpg>, 2012. [Online; accessed Nov 13, 2015].
- [8] CORPORATION, I. D. The digital universe of opportunities: Rich data and the increasing value of the internet of things, 2014.
- [9] COULOURIS, G., DOLLIMORE, J., KINDBERG, T., AND BLAIR, G. *Distributed Systems: Concepts and Design*, 5th ed. Addison-Wesley Publishing Company, USA, 2011.
- [10] HINDMAN, B., KONWINSKI, A., ZAHARIA, M., GHODSI, A., JOSEPH, A. D., KATZ, R., SHENKER, S., AND STOICA, I. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2011), NSDI’11, USENIX Association, pp. 295–308.
- [11] ISARD, M. Autopilot: Automatic data center management. *SIGOPS Oper. Syst. Rev.* 41, 2 (Apr. 2007), 60–67.
- [12] SCHWARZKOPF, M., KONWINSKI, A., ABD-EL-MALEK, M., AND WILKES, J. Omega: flexible, scalable schedulers for large compute clusters. In *SIGOPS European Conference on Computer Systems (EuroSys)* (Prague, Czech Republic, 2013), pp. 351–364.
- [13] TANENBAUM, A. S., AND STEEN, M. v. *Distributed Systems: Principles and Paradigms (2Nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

- [14] VAVILAPALLI, V. K., MURTHY, A. C., DOUGLAS, C., AGARWAL, S., KONAR, M., EVANS, R., GRAVES, T., LOWE, J., SHAH, H., SETH, S., SAHA, B., CURINO, C., O'MALLEY, O., RADIA, S., REED, B., AND BALDESCHWIELER, E. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing* (New York, NY, USA, 2013), SOCC '13, ACM, pp. 5:1–5:16.
- [15] VERMA, A., PEDROSA, L., KORUPOLU, M., OPPENHEIMER, D., TUNE, E., AND WILKES, J. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems* (New York, NY, USA, 2015), EuroSys '15, ACM, pp. 18:1–18:17.
- [16] VERMA, A., PEDROSA, L., KORUPOLU, M. R., OPPENHEIMER, D., TUNE, E., AND WILKES, J. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)* (Bordeaux, France, 2015).