Implentation of an simplistic Interface for Big Data Workload Scheduler in Kubernetes

Implentation of an simplistic Interface for Big Data Workload Scheduler in Kubernetes

Lukas Schwerdtfeger

A thesis submitted to the
Faculty of Electrical Engineering and Computer Science
of the
Technical University of Berlin
in partial fulfillment of the requirements for the degree
Bachelor Technische Informatik

Berlin, Germany
December 22, 2015

Main supervisor:

Prof. Dr. habil. Odej Kao, Technical University of Berlin

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

Berlin, den

Zusammenfassung

Kurze Zusammenfassung der Arbeit in 250 Wörtern.

Abstract

Short version of the thesis in 250 words.

Acknowledgements

This chapter is optional. First of all, I would like to...

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

The current population is producing more and more data. This creates an excellent opportunity for many businesses. Businesses willing to profit from collected data by using it to improve their sales strategies have to collect and store GigaBytes and upwards to ExaBytes of data. With storage costs becoming more affordable, companies are even less likely to toss away potential valuable data, creating so-called Data Lakes.

Collecting data is only the first step. It takes many stages of processing, through aggregation and filtering, to extract any meaningful information. Usually, the sheer mass of collected data makes it not very useful, to begin with.

Unfortunately, when working with ExaBytes of data, it is no longer feasible to work on a single machine. Especially when dealing with a stream of data produced by a production system and the information collected from yesterday's data is required the next day, or ideally immediately.

Scaling a single machine's resources to meet the demand is also not feasible. It is either very expansive or might just straight up not be possible. On the other hand, cheap commodity hardware allows the scaling of resources across multiple machines is much cheaper than investing in high-end hardware or even supercomputers.

The complexity of dealing with a distributed system can be reduced using the abstraction of a cluster. A **Cluster Resource Manager** is used, where a system of multiple machines forms a single coherent cluster that can be given tasks to.

Stream Processing of Data across such a cluster can carry out using Stream or Batch Processing Frameworks, such as Apache Spark or Apache Flink. These Frameworks already implement the quirks of dealing with distributed systems and thus hide the complexity.

The problem is that multiple Batch Jobs running on a single cluster need resources that need to be allocated across the cluster. While Resource Allocation is the Task of the Cluster Resource Manager, the manager usually does not know how to allocate its resources optimal and often requires user *(TODO: 1)* to specify the resources that should be allocated per job. This usually leads to either too little resources being allocated per job, starving jobs and increasing the runtime, or more often over-committing resources and thus leaving resources in the cluster

unused.

Another problem that arises is the fact that even though the reoccurring nature of Batch Jobs, not all BatchJobs use the same amount of Resources. Some are more computationally intensive and require more time on the CPU, while others are more memory intensive and require more or faster access to the machine's memory. Others are heavy on the I/O usage and use most of the system's disk or network devices. This shows up in vastly different Job runtime (also total runtime) depending on the Scheduling of Batch Jobs across the Cluster.

Finding an intelligent Scheduling Algorithm that can identify reoccurring Jobs and estimate their resource usage based on collected Metrics and thus create optimal scheduling is not an easy task. It also requires a lot of setup when dealing with a Cluster Resource Manager.

1.1.0.1 TODO:

1. not just a user but the cluster user which is submitting the job

1.1.0.2 Open:

◇ How much detail is required here?

1.2 Problem Description

(TODO: Cluster Resource Manager, like YARN, were focused around Batch-Application because they existed because of Apache Hadoop/Map-Reduce ecosystem)

Cluster Resource Managers, like YARN, emerging from Apache Hadoop, were centered around Batch Frameworks.

With the rise of Cloud Computing and all the benefits that come with it, companies were quick to adopt new cloud computing concepts. The Concept of a Cluster Resource Manager introduced a notion of simplicity to those developing applications for the cloud. A Cluster Resource Manager now managed many aspects that used to be handled by dedicated Operations-Teams.

Kubernetes, a Cluster Resource Manager that was initially developed by Google, after years of internal use, provided an all-around approach to Cluster Resource Management for not just Batch-Application. The global adoption of Kubernetes by many leading companies, led to the growth of the ecosystem around it. Kubernetes has grown a lot since and has become the new industry standard, benefiting from a vast community.

Old Batch-Application-focused Cluster Resource Managers that used to be the industry standard are being pushed away by Kubernetes. Unfortunately, vastly different Interfaces or Scheduling Mechanism between other Cluster Resource Managers usually block the continuation of existing research done in the field of Batch Scheduling Algorithms.

Finding an efficient scheduling algorithm is a complex topic in itself. Usually, the setup required to further research existing scheduling algorithms is substantial. Dealing with different Cluster Resource Manager further complicates continuing on already existing work.

1.2.0.1 TODO:

1.2.0.2 Open:

- ◇ This sections contains a lot of text that may be better suited to the introduction section, but i don't really now what else to put in here
- ◇ Not happy with the ending of this chapter, like introduction it's really only one paragraph at the end that explains the intended contribution of this work

1.3 Goal of this Thesis

To aid further research in the topic of Batch-Scheduling-Algorithms, the goal of this thesis is to provide a simplistic interface for Batch-Scheduling on Kubernetes.

(TODO: Explain how already existing Schedulers like Mary and Hugo do not run on Kubernetes due to different interface/interactions)

Already existing Scheduling Algorithms, like Mary and Hugo, were initially developed for the Cluster Resource Manager YARN. Reusing existing Scheduling Algorithms on the nowadays broadly adopted Cluster Resource Manager Kubernetes is not a trivial task due to vastly different interfaces and interaction with the Cluster Manager.

(TODO: Explain why the Setup of Kubernetes has become easier: Cloud Providers, MiniKube)

Extending existing research to the more popular Resource Manager Kubernetes provides multiple benefits.

1. Research on Scheduling Algorithms for YARN has become less valuable due to less usage
2. The large ecosystem around Kubernetes allows for a better development environment due to debugging and diagnostic tooling
3. Initial setup of a Kubernetes cluster has become smaller due to applications like MiniKube, which allows a quick setup of a cluster in the local machine and Cloud Providers offering Kubernetes Clusters as a service.

(TODO: Describe the Interface here)

The interface should provide easy access to the Kubernetes Cluster, allowing an external scheduler to place enqueued Batch-Jobs in predefined slots inside the cluster.

For an external scheduler to form a scheduling decision, the interface should provide an overview of the current cluster situation containing:

1. Information about empty or in use slots in the cluster
2. Information about Jobs in the Queue
3. Information about the history of reoccurring Jobs, like runtime

It should be possible for an external scheduler to form a scheduling decision based on a queue of jobs and metrics collected from the cluster. The interface should accept the scheduling decision and translate it into Kubernetes concepts to establish the desired scheduling in the cluster.

(TODO: Explain shortcomings of Kubernetes)

Currently, the Kubernetes Cluster Resource Manager does not offer the concept of a Queue. Submitting jobs to the cluster would either allocate resources immediately or produce an error due to missing resources.

Kubernetes does not offer the concept of dedicated Slots for Applications either. While there are various mechanisms to influence the placement of specific applications on specific nodes, these might become unreliable on a busy cluster and require a deep understanding of Kubernetes concepts, thus creating a barrier for future research.

1.3.0.1 TODO:

- ◇ Implementation of easy to use Interface that would allow already Batch Job Scheduling Algorithms likes Hugo and Mary to be run with small changes, on the popular Cluster Management Software Kubernetes

1.3.0.2 OPEN:

- ◇ Use of “should”. Okay? or Rather what it does?

1.4 Structure of this Thesis

The structure of this thesis allows the reader to read it in any order. To guide the reader through this thesis, the structure of this Thesis section will briefly explain which section contains which information.

The Background Chapter is supposed to give a brief overview of this thesis’s underlying concepts. This chapter introduces Big Data Streaming Processing, the Cluster Resource Manager Kubernetes, and Scheduling.

Following the Background chapter, the thesis provides an overview of the approach taken to tackle the problem described in the Problem Description Section. The Approach Section focuses on more profound concepts of Kubernetes and the Scheduling Cycle of the Kubernetes Scheduler. It summarizes the Kubernetes Operator pattern, which is commonly used to extend Kubernetes.

Implementation details will be given inside the Implementation Chapter, where an architectural overview and interaction between individual components are explained. The Implementation section also emphasizes the design Process for the Interface, which is exposed to an external scheduler. A significant part of the implementation is the Operator, which will be discussed extensively. The Implementation chapter shows how the points made inside the Approach Chapter are were implemented in the end. Finally, as the Goal of this Thesis section describes, changes that had to be made to already existing Scheduling Algorithm Implementations are disclosed and discussed.

(TODO: Hard to describe what is going to happen inside the Evaluation, if i don’t have anything to evaluate yet)

An Evaluation of the research and contribution done by this thesis will be presented inside the evaluation chapter. Here its functionality is demonstrated. This section will also outline some of the limitations.

Before concluding the thesis, a comparison between State of the Art Technology for Kubernetes and Non-Kubernetes Scheduling Frameworks is made.

1.4.0.1 TODO:

- ◇ Thesis starts by giving a brief background to Big Data Streaming Processing, Cluster Management Systems (Kubernetes), and Scheduling
- ◇ Discuss the Approach this thesis takes on tackling the Problem Description, by explaining how scheduling in Kubernetes works and what it takes to Extend Kubernetes (using the Operator Pattern)
- ◇ Implementation Details that a worth mentioning:
  - ○ An architectural Overview.
  - ○ The Process of designing an Interface
  - ○ The Operator that is used to extend Kubernetes
  - ○ Changes that had to be made to existing Algorithms (and their tests)
- ◇ How the work of thesis is evaluated, by testing it's functionality, comparing results from previous work and finally outlining its limitations
- ◇ Comparing the Work that was done to current State of the Art Technology like the Batch Scheduling Framework Volcano and comparing to Scheduling approaches that are not available on Kubernetes
- ◇ A final Conclusion, with a note on future work, that is missing from the current implementation or requires rethinking.

2
Background

2.1 Big Data Stream Processing

Big Data Processing aims to solve the problem of analyzing large quantities of data. In the last years, the amount of data that is being generated has exploded. This creates a Problem where single machines can no longer analyze the data in a meaningful time. While the Big Data Processing frameworks still work on single machines, computation is usually distributed across many processes running on hundreds of machines to analyze the data in an acceptable time.

Analyzing data on a single Machine is usually limited by the resources available on a single machine. Unfortunately, increasing the resources of a single machine is either not feasible from a cost standpoint or simply impossible. There is only a limited amount of Processor time, Memory, and IO available. Cheap commodity hardware allows a cluster to bypass the limitations of a single machine, scaling to a point where the cluster can keep up with the generated data and once again analyze data in a meaningful time frame.

Dealing with distributed systems is a complex topic in itself. Many assumptions that could be made in a single process context are no longer valid. Scaling to more machines increases the probability of failures. Distributed Systems need to be designed to be resilient against Hardware-Failures, Network Outages/Partitions and are expected to recover from said Failures. Having a single failure resulting in no or an invalid result will not scale to systems of hundreds of machines, where it is unlikely not to encounter a single failure during execution.

Big Data Processing Frameworks can be put into two categories, although many fall in both categories. Batch Processing and Stream Processing. In Batch Processing, data size is usually known in advance, whereas Stream Processing expects new data to be streamed in from different sources during the Runtime. Batch Processing Jobs will complete their calculation eventually, and Stream Processing, on the other hand, can run for an infinite time frame.

(TODO: DAG, Images) Internally, Big Data Processing Frameworks build a directed acyclic graph (DAG) of Stages required for the analysis. Stages are critical for saving intermediate results, to resume after failure, and are usually steps during the analysis, where data moving across processes is required. Stages can be generalized in Map and Reduce Operations. Map operations can be performed on many machines in parallel, without further knowledge about

the complete data sets, like extracting the username from an Application-Log-Line. Reduce Operations require moving data around the cluster. These are usually used to aggregate data, like grouping by a key, summing, or counting.

(TODO: Synchronization)

Partitioning of the Data is required due to the limitations of each single Machine. Datasets that the Distributed Processing Frameworks analyze are usually in the range of TeraBytes which is multiple magnitudes higher than the amount of Memory that each Machine has available. *(TODO: Distributed Data Store like HDFS)* While Persistent Memory Storage, like Hard-Drives, might be closer to the extent of BigData, Computation will quickly become limited by the Amount of I/O a single machine can perform.

The user of Big Data Processing frameworks is usually not required to think about how an efficient partition of the data across many processes may be accomplished. Frameworks are designed in a way where they can efficiently distribute a large amount of work across many machines.

2.1.0.1 TODO:

- ◇ Explain why cluster computing is required to deal with the Big Data Problem
- ◇ Explain what makes distributing computation across a cluster hard
- ◇ Explain the Value of already existing Big Data Stream Frameworks like Spark and Flink
- ◇ Explain on a high level how these work
  - ○ Explaining the DAG is required in order to later differentiate between DAG-level scheduling and “Pod”-Level scheduling
  - ○ Driver and Executor Pods
- ◇ Mention the use on Kubernetes using the Spark and Flink Operator

2.1.0.2 Input

- ◇ mehr generisch
- ◇ unterschied stream/batch

2.2 Scheduling

In general, scheduling is the process of assigning resources to a task. This includes the question:

1. Should any resources be allocated for the task at all
2. At which point in time should resource be allocated
3. How many resources should be allocated
4. Which of the available resources should be allocated

(TODO: Where scheduling is necessary)

Scheduling is essential for Operating Systems that need to decide which process should get CPU time and which processes may need to wait to continue computation. In the case of multiple CPU, a decision has to be made which CPU should carry out the computation. The Operating System is not just concerned with CPU-Resources, but also I/O Device resources.

Some devices may not work under concurrent usage and require synchronization. Who is allowed to access it?

(TODO: What scheduling policies/strategies exist, and what are they aiming to optimize)

In some cases, a simple FIFO scheduling that works on tasks in order there were submitted produces acceptable results. *(TODO: Scheduling is used to optimize for Deadlines/Throughput/FastRespons* Scheduling depends on a goal. Some Algorithms aim to find the optimal schedule to respect any given deadlines. Whereas some distinguish between Soft and Hard Deadlines, where ideally you would not want to miss any deadlines, occasionally missing Soft deadlines to guarantee Hard Deadlines are met is acceptable. In general, finding a single best schedule that allows resources to be allocated optimally is not possible. Scheduling for a fast response time or throughput might prefer shorter tasks to be run, when possible, and might starve longer running tasks for a long time before progress can be made.

(TODO: Preemptive)

A Scheduling algorithm might allow preemption, where the currently active task could be preempted for another task to become active. Some scheduling algorithms account for the potential overhead of preempting the current task (like a context switch).

The higher up the Stack *(TODO: stack = single process -> os -> vms -> Distributed Systems)*, more and more potential schedules become possible. It seems a wise choice for the scheduling to be handled in their respective stack layers.

The Question of Scheduling in a Distributed System is now the question of which machines resources should be used for which task. Here Scheduling algorithms need to pay attention to the characteristics of a Distributed System: 1. Potential heterogeneity of the system, with machines of different Hardware and different Operating Systems or Software 2. Spontaneously adding and removing resources of the Cluster 3. Interference between Applications residing on the same machine, same rack, same network switch, etc. (CO-Location)

While some of these factors can be controlled, different algorithms can be chosen for various use cases.

*(TODO: Scheduling in Stream Processing: DAG Level Scheduling / Container Level Scheduling)

In Stream-Processing, we deal with a multitude of different levels of scheduling. Stream Processing Frameworks build the DAG based on the job submitted. The initial DAG breaks done the job into their respective Map and Reduce Operations. These Operations will be broken down into smaller Tasks based on the Partitioning of Data. Finally, the Tasks may be executed on an arbitrary number of machines (technically not machines, but processes). Optimizing the schedule of tasks to a machine will be called DAG-Level scheduling and may now also include factors like Data-Locality.

(TODO: Scheduling on the Cluster Level: The Interesting Topic of this Thesis)

Moving Up one level Higher in the Stack, we are concerned with running multiple Jobs inside the same cluster, and a decision needs to be made which job can spawn their executor on which nodes. Executors are packaged on Containers. The containers are isolated, so they can not access each other. Unfortunately, isolating processes from each other in a container forces the underlying machine to need to know how much of the system's CPU and memory each container should have.

2.2.0.1 TODO:

- ◇ Explain what Scheduling is
- ◇ Different kind of scheduling
  - ○ DAG Scheduling done by Spark (Not what this thesis is about)
  - ○ POD Scheduling done by the Cluster Resource Manager
    - ∗ Co-Location
    - ∗ Packing
- ◇ Explain why Scheduling is Important
  - ○ Co-Location Problem
  - ○ Low Resource Usage (Graph from Google)
  - ○ Results from Hugo/Mary Paper

2.2.0.2 Open:

- ◇ Should maybe start a bit less specific about this Thesis and find more Information about Scheduling in general or is it fine if the Background section starts of general and tailors towards the topic of my thesis?

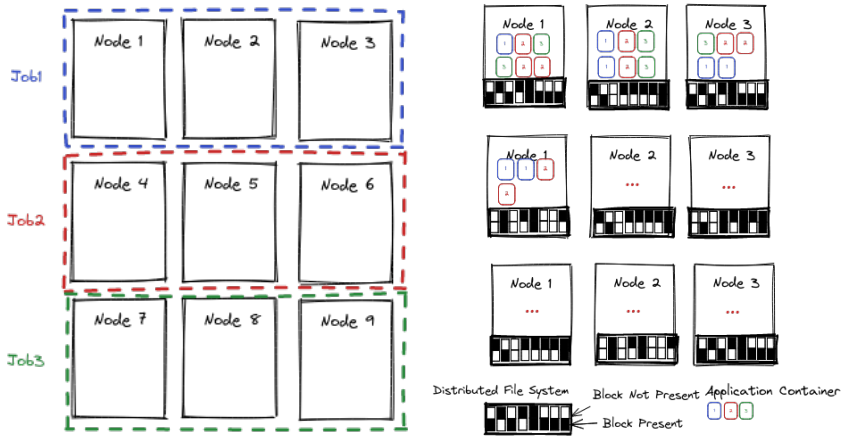
2.3 Cluster Management Systems

- ◇ Abstraction for Systems with many machines

(TODO: Super Computer/HPC) - Super Computer with high-end hardware - Scale Work across multiple machines benefitting from cheap commodity hardware - Coarse Grain: Static partitioning of Machines across different workloads is enough

(What is MapReduce, maybe put into Big Data Stream Processing)

- ◇ Hadoop is one of Many Open-Source MapReduce implementations *(TODO: Frameworks)* Cluster computing using commodity hardware was driven by the need to keep up with the explosion of data. The Initial Version of Hadoop was focused on Running MapReduce Jobs to process a Web Crawl (*YARN Paper*). Despite the initial focus, Hadoop was widely adopted evolved to a state where it was no longer used with its initial target in mind. Wide adoptions have shown some of the weaknesses in Hadoops architecture:
- ◇ Tight Coupling between the MapReduce Programming model and Cluster Management
- ◇ Centralized Handling of Jobs will prevent Hadoop from Scaling



The tight coupling leads Hadoop users with different applications to abuse the MapReduce Programming model to benefit from cluster management and be left with a suboptimal solution. A typical pattern was to submit 'map-only' jobs that act as arbitrary software running on top of the resource manager. [9] The other solution was to create new Frameworks. This caused the invention of many frameworks that aim to solve distributed computation on a cluster for many different kinds of applications [8]. Frameworks tend to be strongly tailored to simplify solving specific problems on a cluster of computers and thus speed up the exploration of data. It was expected for many more frameworks to be created, as none of them will offer an optimal solution for all applications.[8]

Initially, Frameworks like MapReduce created and managed the cluster, which only allowed a single Application across many machines—running only a single application across a cluster of nodes led to underutilization of the cluster's resources. The Next generation of Hadoop allowed it to build ad-hoc clusters, using Torque [6] and Maui, on a shared pool of hardware. Hadoop on Demand (HoD) allowed users to submit their jobs to the cluster, estimating how many machines are required to fulfill the job. Torque would enqueue the job and reserve enough machines once they become available. Torque/Maui would then start the Hadoop Master and the Slaves, subsequently spawn the Task and JobTracker that make up a MapReduce Application. Once all Tasks are finished, the acquired machines are released back to the shared resource pool. This can create potentially wasteful scenarios, where only a single Reduce Task is left, but many machines are still reserved for the cluster. Usually, Resources requirements are overestimated and thus leaves many clusters resources unused.

With the Introduction of Hadoop 3, the MapReduce Framework was split into the dedicated Resource Manager YARN and MapReduce itself. Now MapReduce was no longer running across a Cluster, but it was running on top of YARN, who manages the cluster beneath. This allows MapReduce to run multiple Jobs across the same YARN Cluster, but more importantly, it also allows other Frameworks to run on top of YARN. This moved a lot of complexity away from MapReduce and allowed the framework to only specialize on the MapReduce Programming Model rather than managing a cluster. This finally allows different Programming Models for Machine Learning tasks that tend to perform worse on the MapReduce Programming model [10] to run on top of the same cluster as other MapReduce Jobs.

Using the ResourceManager YARN allows for a more fine-grain partitioning of the cluster resources. Previously a static partitioning of the clusters resources was done to ensure that a specific application could use a particular number of machines. Many applications can significantly benefit from being scaled out across the cluster rather than being limited to only a few

machines. - Fault tolerance: Frameworks use replication to be resilient in the case of machine failure. Having many of the replicas across a small number of machines defeats the purpose - Resource Utilization: Frameworks can scale dynamically and thus use resources that are not currently used by other Applications. This allows applications to scale out rapidly once new nodes become available - Data Locality: Usually, the data to work on is shared across a cluster. Many applications are likely to work on the same set of data. Having applications sitting on only a few nodes, but the data to be shared across the complete cluster leads to a lousy data locality. Many unnecessary I/O needs to be performed to move data across the cluster.

Fine-grain partitioning can be achieved using containerization, where previously applications were deployed in VMs, which would host a full Operating System. Many containers can be deployed on the same VM (or physical machine) and share the same Operating System Kernel. The hosting Operating makes sure Applications running inside containers are isolated by limiting their resources and access to the underlying Operating System.

Before Hadoop 3 with YARN was published, an Alternative Cluster Manager Mesos was publicized. Like YARN, Mesos allowed a more fine granular sharing of resources using containerization. The key difference between YARN and Mesos is how resources are scheduled to frameworks running inside the cluster. YARN offers a resource request mechanism, where applications can request the resources they want to use (*TODO: fact check*), and Mesos, on the other hand, offers frameworks resources that they can use. This allows frameworks to decide better which of the resources may be more beneficial. This enables Mesos to pass the resource-intensive task of online scheduling to the frameworks and improve its scalability.

(TODO: Kubernetes) Kubernetes was initially developed by Google and released after multiple years of intern usage. Kubernetes was quickly adopted and has become the defacto standard for managing a cluster of machines. Kubernetes offers a descriptive way of managing resources in a cluster where manifests describing the cluster's desired state are stored inside a distributed key-value store etcd. Controllers are running inside the cluster to monitor these manifests and do the required actions to bring the cluster into the desired state. Working with manifest abstracts away many of the problems that arise when deploying Applications to a cluster. Usually, an Operations Team was required to manage applications across the cluster. With Kubernetes offering the required building blocks and the mechanism of a control-loop, the operator pattern in combination with Custom Resource Definitions is commonly used to extend Kubernetes functionalities.

2.3.0.1 Notes (Ignore):

- ◇ HPC Super Computer / Resources
- ◇ Kubernetes Section is definitely not complete
- ◇ Explain what a Cluster Resource Manager is doing
  - ○ Abstraction of using a single cluster as a single Machine
  - ○ Managing given resources making it scalable by adding more machines
- ◇ Show what are the differences between YARN and Kubernetes
  - ○ YARN: Emerging from Hadoop was design to Work with Batch Applications

- Kubernetes: All Round Cluster Manager, with a Big Community

(TODO: Cloud Computing) - Coarse Grain approach is no longer feasible - Applications may scale up or down, for and a partitioning of the cluster has to be done dynamically - Containerization, allows to run many different applications on the same machine without much overhead, like creating a new virtual machine - Different Applications may share same Node to increase overall utilizations of resources - Applications benefit from data locality, where creating application on a node that already contains the data, will not use additional I/O Resources - Different Applications are likely to work on the same data

Mesos and Kubernetes - etcd vs ZooKeeper - Mesos Resource Offer leave scheduling to Framework - Kubernetes follows a descriptiv approach, where the user of the cluster describes a state that the cluster should be in and the resource manager exectues the required actions. This makes Kubernetes very extensible. - Kubernetes builds a virtual network across nodes - Kubernetes general purpose

YARN Paper - Hadoop thightly focused on Web Crawling for Hadoop at Yahoo! - Broad Adoption streched intial focus - Thight Coupling of MapReduce Programming Model and Resource Manager - Centralized handling of Jobs control flow from the JobTracker

- MapReduce Programming model was abused for other purposes then it was initially design
- Common Pattern of Map-Only that was used for Forking?-Web-Services and Gang-Scheduled Computation. In general Developer came up with clever solutions to run all kinds of software on top of hadoop
- Misuse and Broad adoption exposed many substantial issues with Hadoops archicture and implementation
- YARN moves Hadoop pasts its original incarnation, by breaking up the monolithic architecture of Hadoop, splitting it into a Resource Manager from the programming model
- YARN delegates many scheduling-related functions to per-job components
- This makes MapReduce just one of many applications that can run on top of YARN
- Allows for great choice of Programming Models using different Frameworks
- Programming Frameworks running on top of YARN can manage their intra-application communication, execution flow and dynamic optimization by them self, unlock performance benefits

2.3.0.2 Historical

- Yahoo! WebMap
- Needs to be scalable
- Ad-hoc clusters
- Initially user bring up a handful of nodes, load their data into HDFS, write a MapReduce job, then tear it down
- Hadoop becomes more fault tolerant and persistent HDFS would become the norm
- Operators just upload potential interesting data into the the HDFS, attracting analysts. –> Mulit-Tenancy of HDFS
- To address Multi Tenancy HoD was deployed, using Torque and Maui to allocate Hadoop Cluster on a shared pool of Hardware

- ◇ User submit Job, with required amount of rescoures to torque, which then waits until enough resources are avaible
- ◇ Torque would start the Hadoop Leader, which then interacts with Torque and Maui to create the Hadoop Slaves, which will then Spawn the TaskTracker + JobTracker
- ◇ Job Tracker would then accept a series of Jobs
- ◇ User can then release the Cluster. The System will then return logs to the user and return nodes to the cluster
- ◇ HoD allows slightly older version of Hadoop to be used
- ◇ HoD shortcomming:
- ◇ Torque did not account for data locality
- ◇ Bad resource utilization, last task of job may block hundreds of machines, Cluster were not resized between jobs, Users usually overestimate the number of nodes required
- ◇ Shared Cluster
- ◇ Resource Granularity was to coarse
- ◇ JobTracker does not scale as HDFS, JobTracker failure was a complete failure of all jobs
- ◇

3
Approach

3.1 Scheduling in Kubernetes

In this section, the Scheduling Model of Kubernetes will be introduced as it is a vital part of the implementation for the Interface.

The smallest unit of *deployment* in Kubernetes is a Pod[4]. While a pod may consist have multiple containers. Containers in a pod are guaranteed to run on the same node. Containers in Pods also share storage and network resources across the container boundary. In General, containers inside the same Pod are tightly coupled and commonly used in a sidecar pattern to extend the main container with common functionalities across the cluster, like the Kube-RBAC-Proxy[1] that is frequently used with Containers that interact with the Kubernetes API and require authorization.

Usually, in Kubernetes, Pods are not created by themself but are managed by resources that build on top of them. Most commonly, Pods are used in Combination with Jobs, Deployments, or Statefulsets, which control the Lifecycle of the Pod.

The scheduling Problem in Kubernetes is the problem of deciding which pods are running on which node. For some pods, the question can be easily solved. For example, Pods that a DaemonSet controls are by the specification of the DaemonSet running on every node. Without further information, a feasible choice of scheduling pods onto nodes seems to be simple round-robin scheduling. Every Pod that requires scheduling gets scheduled onto the next nodes until all nodes have pods then the cycle is repeated. However, both pods and nodes can influence the scheduling.

Pods can specify the resources they are going to use and may even set a Hard Requirement in the form of a request for resources they require to run. Scheduling needs to take the resources requests into account when scheduling a pod across nodes. Pods can also directly influence the node they should be scheduled on, either through specifying a *nodeName* directly, a *nodeSelector*, which identifies possible multiple nodes, or a more general concept of *affinity*. Affinities provide the ability to set Hard and Soft Requirements, where a Pod may become unschedulable if a Hard Requirement cannot be met, and a Soft requirement is not preferred but an acceptable decision. With Affinities, even inter-pod affinities can be specified where pods can choose not to be scheduled on a node where another pod is already deployed.

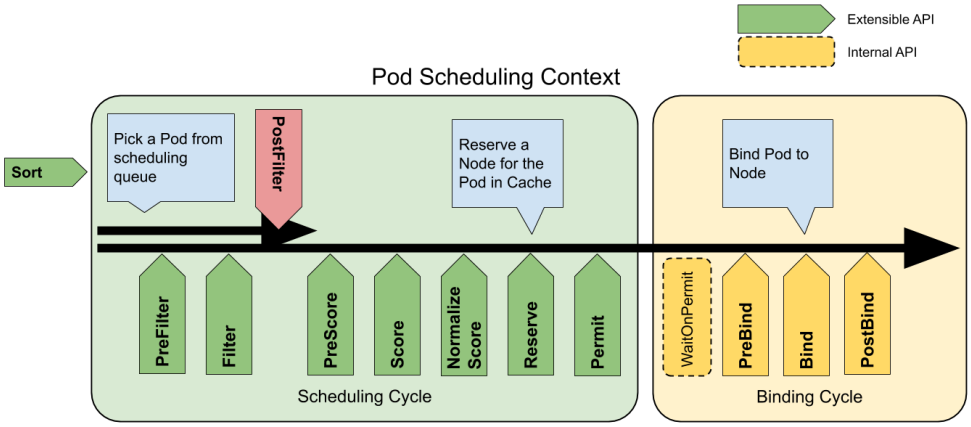


Figure 3.1: Scheduling Cycle

On the other side of the scheduling equation, nodes can also specify so-called Taints, where only Pods with the fitting toleration can be scheduled onto the node. Kubernetes also use the Taint mechanism to taint nodes that are affected by Network problems, Memory-Pressure, or are not ready after a restart. The taint specifies if just new pods cannot be scheduled *NoSchedule* onto the node, or if even already running pods should be evicted *NoExecute*. These nodes with these taints should usually not be chosen during scheduling.

However, Pods can bypass the Taint node if they have the fitting toleration. For example, if a node becomes unreachable due to a network outage, nodes affected will be tainted with the *"node.kubernetes.io/unreachable"* taint, which is set to *NoExecute*. Without the fitting toleration, Pods can not be scheduled onto the unreachable nodes, but they may still keep executing while running on an unreachable node. This is possible because Kubernetes by default creates the unreachable toleration for all pods (unless specified otherwise). However, the toleration set by Kubernetes also specifies a duration where the taint is tolerated (by default, 5 minutes).

Quick Side Note: Taints and Tolerations, and Pods and Nodes, here all refer to the resources stored inside etcd. If a physical node becomes unreachable, the Resource is updated with the Manifest. If the physical containers are still running or cannot be determined, but for the duration of the toleration, they do count as running and are not replaced by new Pods.

3.1.1 Scheduling Cycle

Kubernetes Scheduling is based on the Kubernetes Scheduling Framework. As the extensible nature of Kubernetes, the Scheduling framework is also extensible. Customization to the scheduling can be done at so-called Extension Points.

Each attempt of scheduling pods to nodes is split into two phases, the *Scheduling Cycle*, and the *Binding Cycle*.

The Scheduling Cycle is more interesting in the context of this work. In this part of the cycle, the decision is made on which node to schedule which Pod. The Binding Cycle is the scheduling phase, where the Pod is being deployed to the physical machine. Multiple Scheduling Cycles will not run concurrently, as this would require synchronization between multiple Scheduling Cycles. Since the Scheduling Cycle is relatively fast, executing multiple Scheduling Cycles in parallel seems unnecessary. However, the Binding Cycle requiring the

deployment of Pods is in comparison to the Scheduling Cycle rather long-living and thus may be executed in parallel. Both the Scheduling Cycle and the Binding Cycle have in common is that both may abort the scheduling of a pod in case it may be unschedulable.

As mentioned earlier, the scheduling context can be modified through plugins that intern use the Scheduling Framework's extension points. The Pod Scheduling Cycle can be broken into a Filter, Scoring Reservation and Permit State. Multiple Plugins can be registered for any of the different extension points and will run in order.

However, the first Extension point is the Queue Sort Extension point. Sometimes multiple pods require a scheduling decision. The Queue Sort Extension point extends the Scheduling Cycle with a comparison function that allows the Queue of Pods to wait for a scheduling decision to be sorted. Usually, multiple Plugins can influence the scheduling decision at a time; however, having numerous Plugins sorting the queue of waiting pods will not result in anything meaningful. Thus, only a single plugin can control the queue at a time.

The following way to influence the Scheduling decision is done using the PreFilter and Filter extension point. The Filter Extension point filters out Nodes that cannot run the Pod. Most extension points also have a pre-extension point used to prepare information about the Pod. As mentioned earlier, any extension point can return an error indicating that the Pod may be unschedulable.

The Filter extension point also has a PostFilter Extension, which is only called if after the Filtering finds no Node. The PostFilter Extension can then be used to find a possible Scheduling using preemption. Preemption is very important, as it is required to model Pods having a higher priority than other Pods. In the context of this work, preemption is required to guarantee that the scheduling from an external scheduler is correctly applied even if other pods in the cluster exist that are unknown to the external scheduler.

The next available extension point, part of the Scheduling Cycle, is the Scoring point. The filter plugin is concerned with hard requirements that prevent pods from being scheduled. If previous filter plugins deem multiple nodes suited for the Pod to be scheduled on, a decision needs to be made to find the best node. The scoring plugin considers soft requirements and gives nodes a lower or higher score that can or cannot fulfill the soft requirements. Finally, all Scores are normalized between two fixed values.

The reserve phase is used to reserve resources on a node for a given pod. This is required due to the asynchronous transition into the Binding Cycle. Pods that are supposed to be bound to a node invoke reservation to prevent possible race conditions between future Scheduling Cycles. Reservation may fail, in which case the Cycle moves to the unreserve phase, and all previously completed reservations are revoked. Usually, the Reservation extension point is used for applications that will not use the default containerization mechanism of Kubernetes and rely on different binding mechanisms.

The final state of the Scheduling Cycle is the Permit state. The Permit State can ultimately deny or delay the binding of a Pod in a case where binding might not be possible or still requires time.

3.1.2 Extending the Scheduler

Common ways of extending the Scheduler are either to implement a custom Scheduler and to replace the KubeScheduler in the cluster, or to use the *Extender* API that instructs the Scheduler to invoke an external API for its scheduling decision.

The pluggable architecture of the Scheduler allows plugins to extend the scheduling cycle at the extension points. Different Scheduler profiles can use different plugins. Profiles can be created or modified using a *KubeSchedulerConfiguration*. Only Pods that specify the Scheduler Profile using *.spec.schedulerName* are scheduled using the profiles.

The Extender Mechanism describes an external Application listing for HTTP Requests issued during the Scheduling Cycle to influence the scheduling decision. Instructing the Scheduler to use an Extender, in the current Implementation of Kubernetes, is not limited to a specific Profile, but all Profile will use the configured Extender. An Extender can specify the Verbs it supports. Verbs in this context refer to Filtering, Scoring, Preempting, and Binding.

(TODO: Check) Note: This work does not replace the KubeScheduler, but deploys a Second Scheduler to the cluster with a Profile that uses the Extender API. This is done because of the limitation that every scheduling profile will use the Extender, which seems unnecessary for this work.

3.2 Extending Kubernetes using the Operator Pattern

The Operator Pattern is commonly used to extend Kubernetes Functionalities.

The Operator Pattern describes a Control-Loop that listens to the desired State of the Cluster and observes the actual state. If the actual state diverges from the desired state, it does the necessary actions to move the actual state into the desired state.

The Operator pattern is based on the already existing design used by Kubernetes Native Resources like Deployments the *Control-Loop*.

Resources like Jobs, Deployments, and Statefulsets describe Pods' desired state. Essentially Kubernetes has controllers that monitor changes to the Resources Manifest and the current cluster situation. The Control-Loop allows a Deployment consisting of a Pod template with a Replication-Factor to the exact amounts of pods running inside the cluster. If any of the Pods fails, the Deployment Controller creates a new one. But on the Flip-Side, the controller also knows when the Resources Manifest is updated. For example, if the container is updated to a more recent version, all containers need to be replaced with the newer version. Deployment Resources have many policies that dictate how these actions should happen. Usually, restarting all Pods at once would not be desired so that the deployment would allow for Rolling-Upgrades. The controller ensures that new Pods are created first and become ready before old Pods are deleted.

Most of the time, Using just a new Controller is not enough to extend Kubernetes. It usually also requires Custom Resource Definitions. CustomResourceDefinitions (CRD) is the Kubernetes way of defining which *kind* of resources are allowed to exist in the cluster. Having multiple Controllers listing to the same Resource, like a Deployment, makes little sense or could even cause issues. Thus the Combination of a new Resource and a Controller that knows how to handle it creates the Operator Pattern. The term *Operator*is used as the controller is created to replace previously manual work of configuring Kubernetes Native Resources done by an Operator. A common use case for the Operator Pattern is to Control Applications at a Higher level, where previously Multiple Deployments and Services may have been required to operate a Database. The Operator Pattern could reduce that to just a single Manifest containing the meaningful configuration. Operators can thus be created by Experts operating the Software and be used by any Kubernetes Cluster.

(TODO: Best Practices when implementating an Operator)

3.2.1 Custom Resource Definitions

- ◇ Kubernetes resources are managed with a RESTful api, where Resources can be queried, created, updated and deleted
- ◇ CRD creates a new RESTful resource path which is managed by the Kubernetes API server.
- ◇ Each version gets its on api
- ◇ Resources can be namespaced or cluster-scoped
- ◇ Custom resources, require a structural schema
    - ○ Non Empty types: (properties for Objects), (items for arrays)

4

Implementation

4.1 Architecture

In this section, all components that make up the Interface are introduced. Here an Architectural overview is presented, and interactions between components are discussed.

The current implementation of the Scheduler Interface consists of 5 Components that will be introduced in this section but discussed in more detail in the Operator Section.

The five components consist of three Reconciler or Control-Loops, the Batch-Job Reconciler, the Slots Reconciler, and the Scheduling Reconciler. The architecture also uses an Extender and, finally, the External-Scheduler facing Web-API.

The Interface that is visible to an External Scheduler is supposed to be simple and only allows the querying of the current cluster situation, information about previous scheduling, and submission of new schedulings. Further, more concrete information, like node metrics, can be queried from a Metric Provider that is commonly deployed along with the Cluster.

Additionally to the three Reconciler, three Custom Resource Definitions (CRDs) are created.

- ◇ *Batch Job* represents an Abstract Batch Job Application and can store information the external scheduler may want to remember for future invocations
- ◇ *Slots*represent a testbed of guaranteed resources available for a *Scheduling*. A *Slots* Custom Resource is a collection of slots across the clusters node that are referenced in a *Scheduling*
- ◇ *Scheduling*represents the decision done by the External-Scheduler. A *Scheduling* maps multiple *Batch Jobs* to *Slots* available in the Cluster. The *Scheduling* also acts as a Queue and submits Jobs into the slots in order once *Slots* become available.

The Batch Job CR is used to Model a reoccurring Batch Job Application. In order to support both Flink and Spark Applications, an abstract Batch Job CR is chosen that maps the state of application-specific CR (link SparkApplications[2] and FlinkCluster[3]) to a common set of possible States. (Information about possible States and the corresponding State Machine

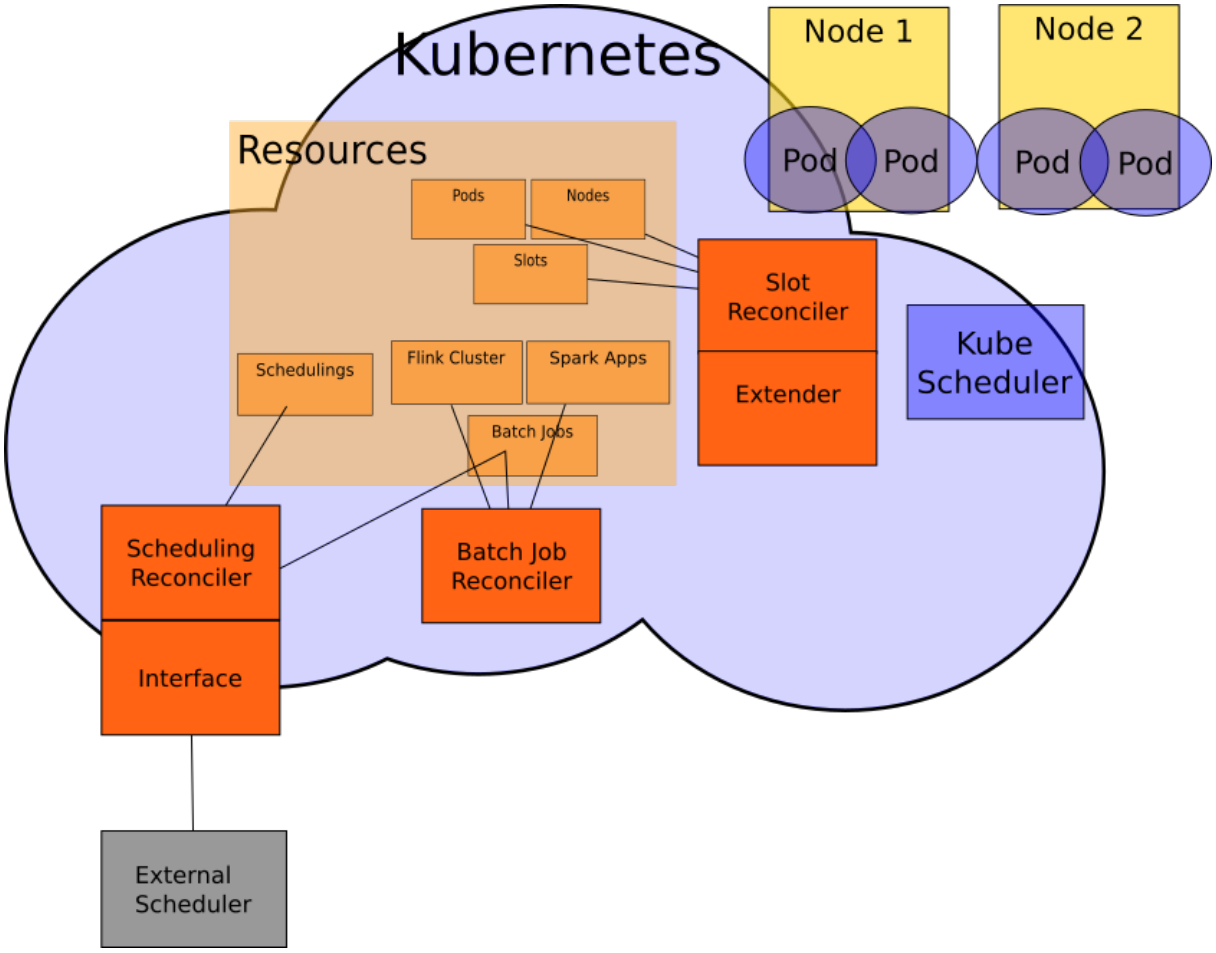


Figure 4.1: Components

are discussed in the Batch Job Operator Section). A Batch Job can be claimed by exactly one *Scheduling*, this is because the Batch Job CR models exactly the life cycle of a single application.

The Slots CR guarantees Resources in the Cluster by creating Ghost Pods, with a specific Resource Request representing the Size of an empty Slot. The Ghost Pods reserve resources by not allowing other Pods requiring scheduling to be scheduled onto the same Node. Using the extender and Preemption, the Slot Reconciler can reserve resources for Pods created by the Batch Job CR.

Finally, the Scheduling CR is passed to the external Interface by the External-Scheduler. Given a Set of BatchJobs and the Slots and the Node they exist on, an External-Scheduler can compute a *Scheduling* that chooses BatchJobs and the Slots they should run in.

Note: Reconciler and Control-Loop can be used interchangeably, but for less confusion with the Spring Boot Concept of a Controller, the principle of a Control-Loop will be called Reconciler**

4.2 Designing the Interface

4.3 Operator

4.3.1 Batch Job Operator

The Batch Job Operator is composed of the Batch Job Reconciler and the Batch Job CRD. The Reconciler is listing for changes, regarding the Batch Jobs CRs and Applications CRs, which are managed by the Spark Operator and the Flink Operator.

The Batch Job Operator knows, given the Batch Job CRs specification, how to construct the corresponding Application. This was made easy due to the fact, that the Batch Job CR is only a thin wrapper around either the Spark or the Flink specification. In addition to the Spark and and Flink CR it also, may contain additional Information, that previous invocations of the External-Scheduler have stored.

Currently the Batch Job CR, does only contain a partial application specific specification. Although it would not matter, if a user would submit a fully specified Flink or Spark Application, the Operator would overwrite, most of the Driver/Executor Pod specific configuration and replication configuration. The Aim of the Batch Job CR is to allow a user to specify only the Required Components, like Application Image containing the actual Application and its arguments like a DateSet or where to find it (e.g. using HDFS).

The BatchJob Reconciler is implemented as a Nested State Machine with anonymous Sub-States. The Approach was chosen, as it creates more comprehensible Software, which can be split easier into Components, and handle Edge Case by Design. More Information about the Design of the State Machine is outlined in the separate Section.

Initially BatchJobs submitted to the Cluster remain in the ReadyState. While in the ReadyState the BatchJob Operator, will not do anything. A *Scheduling* can claim a BatchJob, in which case the BatchJob CR will move into the InQueueState until the *Scheduling* instructs the Batch Job Operator to create the Application and track its lifecycle.

Communication between the Batch Job Reconciler and the Scheduling Reconciler is done via the Kubernetes label mechanism. The *Scheduling* Reconciler places the **ACTIVE_SCHEDULING** labels on any Batch Job CR it will use during the Scheduling. This mechanism ensures that only one Scheduling at a time can use the Batch Job, on the flip side a *Scheduling* can claim multiple *Batch Jobs*. If the active scheduling releases the job, by removing the label, the Batch Job moves back into the Ready State, this can happen could happen at any time, and may even cause any created application to be removed.

Once a Batch Job is in the InQueue State, the reconciler waits for the creation request issued by the *Scheduling* Reconciler. The requests is again done using **APPLICATION_CREATION_REQUEST** labels and specifies desired replication and *Slots* the Application will use.

NOTE: Kubernetes Label mechanism is chosen, because it ensures, that only one scheduling at a time should have control over the Batch Job. A Scheduling will set the Labels on a batch job only if no other has claimed the job. If it detects that it is no longer the active scheduling, which might happen if the same is job claimed by a different scheduling at the same time, it will not proceed the scheduling, but remove all other labels from other jobs to prevent deadlocks. More details how the claiming mechanism was design are in the Scheduling Reconciler Section.

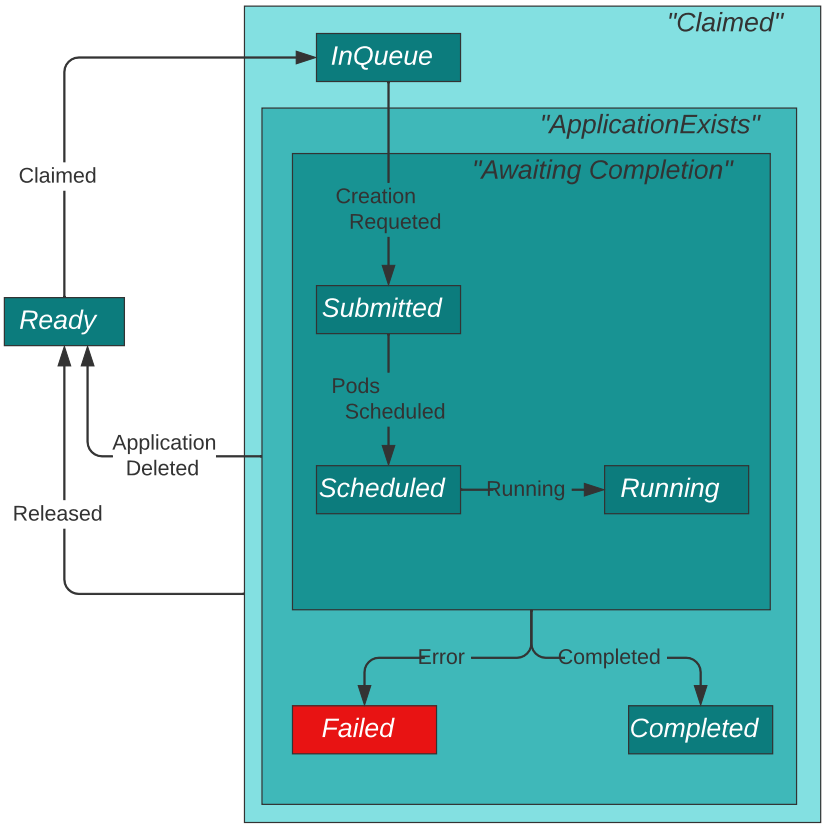


Figure 4.2: StateMachine

When configuring the Application, to be created by the corresponding Operator, there are two types of configuration. Configuration can either be:

- ◇ Persisted inside the Batch Job CR, which is used on every invocation of the Application. This includes the Applications Image and arguments, like the data set
- ◇ Scheduling dependent. These configuration can not be stored inside the CR and need to be supplied with the creation request.

After a Batch Job was requested to create the Application, application specific logic is executed. In any case, the actual steps for deploying the applications to the cluster is done by the Applications Operator (Flink Operator[3] or Spark Operator[2]). The Batch Job Reconciler only instructs the Application Operators, with configurations for the Executor/TaskManager Pods that will be created to be identifiable by the Scheduler Extender.

When creating the application, the following aspects are configures for the Executor/TaskManager Pods:

- ◇ **Resource Requests**: The Container resources are specified by the Testbeds Slots, in order for the pods to fit inside the Slots they need the correct Resource Request. (Currently only CPU and Memory)

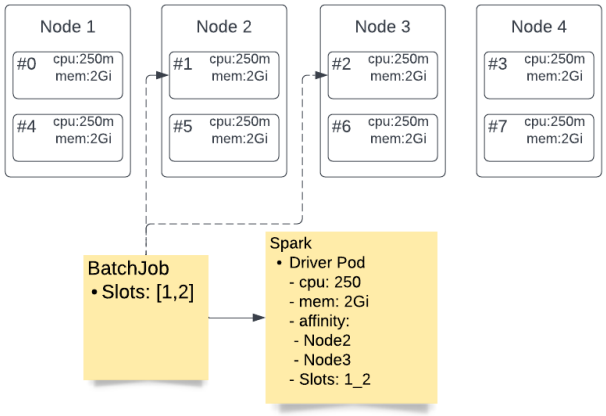


Figure 4.3: Affinities

◇ **Slot IDs**: The scheduling (or the external scheduler) decides which slots are used by which job. In order for the Executor/TaskManager Pods to be placed into the correct slot (technically the correct Node), Pods need to be made identifiable by the Scheduler Extender.

◇ **Replication**: The Number of Executor/TaskManager Pods depends on the Number of Slots that will be used for the Application.

◇ **Priority Class**: Application pods need a *PriorityClass* otherwise, preemption will not be triggered by the Kubernetes Scheduler.

◇ **Scheduler Name**: Application pods need a *SchedulerName* otherwise, the default KubeScheduler will handle the scheduling and thus ignore the Scheduling Extender.

Configuration of **Resource Requests**, **Replication** are straight forward, as both the Spark and Flink Operator expose these via their respective CRDs. The Spark Operator actually, exposes the complete PodSpec for both driver and executor pods, whereas the Flink Operator only exposes a few PodSpec attributes. The Flink Operator had to be extended with the missing configurations. This way **Resource Requests**, **Replication**, **Priority Class**, **Scheduler Name** are configured.

The difference between any of the mentioned above configurations and the **Slot IDs** is that the Application Operators only allow (rightfully so) to specify a single pod spec. The reason for this, is that the Executor/TaskManager Pods are controlled by a Stateful Set, which scales up to the desired Replication. However the above mentioned configurations, are valid for all pods, but *Slot IDs* should be different for all pods.

This issue can be circumvented by the leaving the final decision of which pod goes into which slot to the Extender and submitting a list of SlotIDs to the extender. The Extender can that decide which pod goes into which slot. Pods are configured with a affinity of the combined set of nodes where the slots reside on.

Once the Application was Created the Job moves into the Submission State, and resides there until all Pods of the Application where scheduled, at which point it moves on into the Running State. At this Point the underlying Applications state is monitored, until it moves

into the Application Specific Completed state, (currently for Spark: *Completed* and for Flink: *Stopped*). During the implementation, scenarios were encountered, in which the BatchJob reconciler was not running, and once restarted found Applications in a completed state without passing the scheduling, submission and running state. In Order to prevent any tight coupling none of these transitions are required, to be considered a successful execution.

The BatchJob reconciler tracks the time an application ran, by creating timestamps once the application, started running and its completion. More details about how the Scheduling Extender works are inside the Extender Section.

4.3.2 Slot Operator

The Slot Operator is composed of the Reconciler Loop and the Slots CRD. The Slots CR is supposed to model a Test Bed of slots located in a Cluster of Machines. Slots can have specified Resources. While no application is running inside a slot it is considered *free*. In Order to reserve resources in the cluster, and thus guarantee applications, supposed to be deployed inside a *free* slot to actually get the resources, the Slot Operator needs to:

- ◇ **Reserve Resources** by using so-called Ghost Pods inside the cluster, that specify a resource request and thus reserve the resources
- ◇ **Preempt** Ghost Pods for pods that wants to be deployed inside a slot

The Slots Reconciler listens to changes to the Slots CR, and the current cluster situation. It makes sure that always the correct number of pods, with the specified resource requests, are deployed onto the cluster. The Slots CR is composed of the following configurations:

- ◇ Label Name to Identify any nodes that are part of the Test Bed. Only the Label Name is specified not a specific value. The value is later used to create a distinct order of slots in the cluster.
- ◇ Number of Slots per Node
- ◇ Resource Request per Slot

Given the Test Beds specification, the reconciler listens for all changes to nodes **with** the specified label, but also to all nodes **without** any label in case the label was removed and the test bed needs to be resized. Further it also listens to changes to any pod which is part of the Test Bed.

The typical Reconciliation Loop works as following:

- ◇ Fetch the current cluster situation
- ◇ Calculate the desired cluster situation
- ◇ Find the difference. Either delete undesired Pods, or create desired Pods

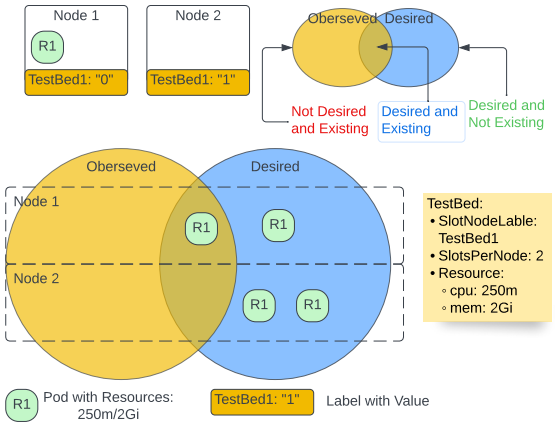


Figure 4.4: New Pods need to be Created

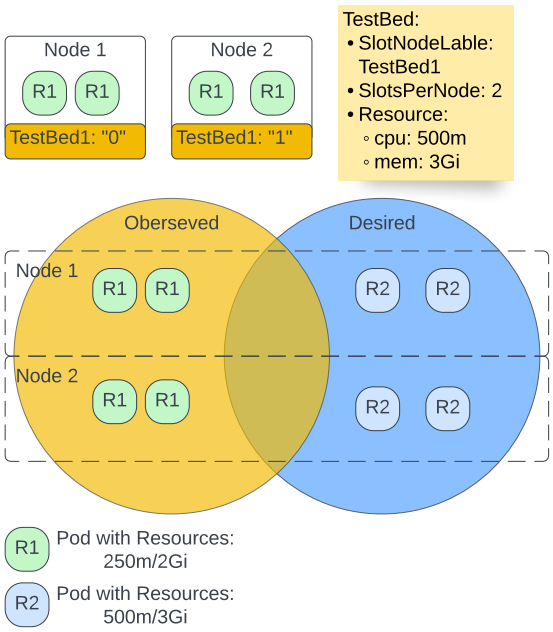


Figure 4.5: ResourcePerSlot Change: New Pods need to be Created

Fetch the current cluster situation, by fetching all pods with the **SLOT** label. Pods are then grouped by their Node, thus creating a list of pods per node. On the Flip side the desired state is calculated by modeling pods for every slot and also group the by node. When comparing pods we consider them equal, if they reside on the same node, have the same resource request, and the same *SlotPositionOnNode*.

Node: The actual position of Slots on a Node does not matter, slots on are Node are only a logical abstraction.

The Reconciler now builds a set of observed pods and a set of desired pods. 4.4 shows the an example scenario where the control-loop realizes, that pods from the desired state are not in the current state, thus creating the missing pods in the *desired and not existing* set. In a different

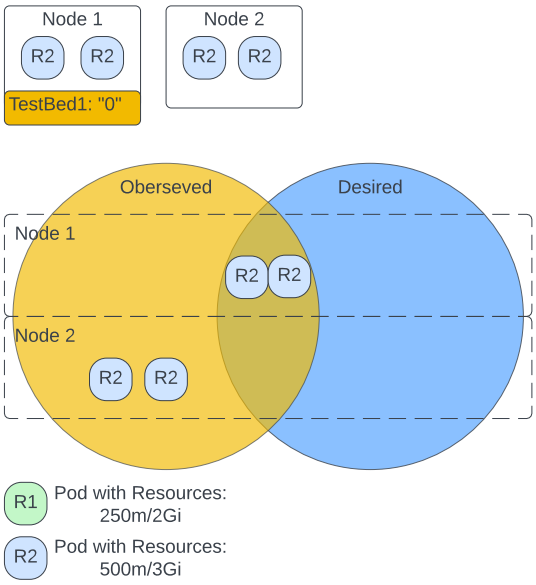


Figure 4.6: Node Change: Pods need to be Deleted

scenario displayed by 4.6 the label on a node was removed, thus reducing the amount of slots in side the testbed. Pods which are in the *existing and not desired* set will be removed. The final set is the *desired and existing* set which contains pods that already have the correct resources requirement and are placed on the correct node.

- ◇ scenario where multiple Pods exists with the same SlotID -> preemption (Excalidraw image)
- ◇ scheduling loop aborts if it detects that cluster is in progress, either detects non scheduled pods, or pods that are terminating

Note: During early implementation it was assumed that there can only be on Test-Bed Active at a Time. While technically that may not be necessary anymore, since every component using the Test Bed specifies its exact name and namespace. Some parts of the Slots reconciler might have been implemented with the initial assumption in mind. E.g. Verifying that enough resources are available on every node, does not account for possible multiple active Test Beds on a node.

Note: Slots are positioned in a round-robin fashion

Currently the SlotOccupationStatus holds the following information:

- ◇ **NodeID** and **NodeName**: which is derived from the Test-Bed Selector Label on the Node
- ◇ **Position**: which is the SlotID,
- ◇ **slotPositionOnNode**: where the Position does unique among the whole Test Bed, Slot-Position on node is only unique per Node
- ◇ **PodName** and **PodUID**: The Name and the Unique Identifier of a Pod that is currently residing inside the Slot
- ◇ **State**: is the current state of the slot, which can either be *free*, *reserved*, or *occupied*

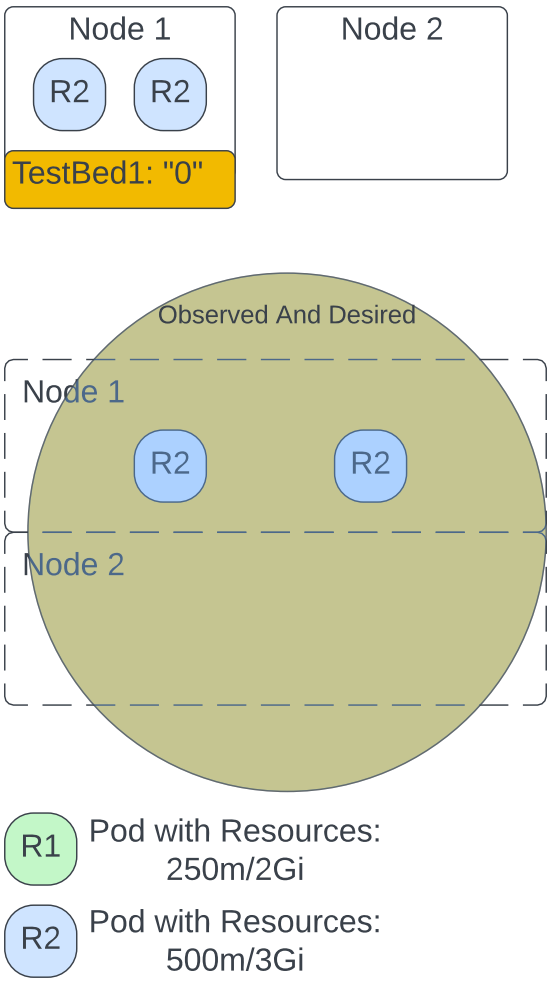


Figure 4.7: Desired State: No Change

4.3.3 Extender

- The Component that interacts with the Kubernetes Scheduler
- Implements both the Filter and the Preemption endpoint
- One of the Problems the extender can solve is setting slot ids per pod, rather than the stateful set or Application CR
- Scheduling configures BatchJob with slots + testbed + replication
- Batch Job configures application
- application somehow creates pods (e.g. using a stateful set which is intern controlled by kubernetes)
- finally pods created by the application are instructed, with the scheduler, which is configured to use the extender
- Scenario1: Slots are sized in a way that does not fit an additional pod of the same size
- Kube Scheduler evaluates resource requests + affinities, filtering does not return any nodes

- ◇ Post filter will trigger preemption, Pods have a priority class -> scheduler believes preemption is useful
- ◇ Kube scheduler calls the extender with an preemption request,
    - ○ scheduler simulates possible preemptions, but aborts once it finds one, this means that the preemption request can be mostly ignored, the only important part is the preemptor pod.
    - ○ pod contains the SlotIDs string which contains all the slots that any of the pods belonging to the application should use
    - ○ extender first checks if any of the slots, within the SlotIds is already reserved for the preemptor pod SlotOccupationStatus, this is only the case for scenario 2.
    - ○ if no reserved pod was found the extender looks for empty slots withIn the SlotIds and reserves it + sets the **SlotID + NonGhostPod** label. Extender now returns the target NodeName + PodUId of the Free Slot that should be preempted.
    - ○ the Kube scheduler, terminates the ghost pod and updates the preemptor pod with the NodeName
- ◇ Scenario2: Slots are sized in a way where an additional pod could fit on the node
- ◇ Kube Schedulers evaluates resource requests + affinities, and finds a valid node
- ◇ Kube Scheduler calls the extenders filter endpoint
- ◇ Filter endpoint calls the preemption method
    - ○ Find an empty Slot on any of the Nodes that passed the Filter Phase
    - ○ However some might not actually be available, in a case where non of the nodes are available the extender returns an empty list of possible nodenames
    - ○ Kube Scheduler will trigger preemption.
- ◇ If slot on node is available the filter result contains the node where the slot resides on and labels are set accordingly on the pod.
- ◇ Pod will be scheduled alongside the ghost pod it is supposed to preempt.
- ◇ Once all pods are scheduled. the Slot Reconciler will no longer abort due to incomplete scheduling, and delete all ghostpods that are preempted by non-ghostpods, with the same slot id.

4.3.4 Scheduler Reconciler

- ◇ Implemented as a state machine
- ◇ Acquire State claims all Batch Jobs and the Test Bed
- ◇ Once all Jobs are in the InQueueState scheduling choses the first n runnable jobs
- ◇ Two Modes: SlotBased + QueueBased (Images)
- ◇ Once Creation was Requested, reconciler waits until all jobs submitted were scheduled.
- ◇ This is required, because the Slot Reservation is not instantaneous. Wait for Batch Job Reconciler + Application Operator until the extender marked slots as reserved.

- ◇ At this point the scheduling waits until slots come available, different Modes require different Condition
- ◇ The Queue Based scheduling, only requires a number of available slots
- ◇ Slot Base scheduling requires specific slots to come available
- ◇ Once the Queue is empty the scheduling moves into the await completion state until all jobs have completed
- ◇ Note: Online scheduling: is possible by updating the scheduling CR and extending the Queue

4.3.5 External Scheduler Interface

- ◇ Interaction with the Scheduling Interface is naturally done via the Kubernetes API, creating, updating, deleting CRs.
- ◇ If the external-scheduler chooses not to directly interact with kubernetes, a thin layer in form of web api is provided.
- ◇ The interface aims to abstract away some of the Kubernetes features like namespaces.
- ◇ The interface allows to create update and delete schedulings. Query for jobs inside the cluster. Query for slots inside the Cluster.
- ◇ The interface contains a web socket server that broadcasts changes to jobs, schedulings, testbed

4.4 Changes to existing Algorithm

5
Evaluation

5.1 Testing

5.2 Comparing to baseline Runtime

5.3 Limitations

5.4 Discussion

6
State of the Art

6.1 Volcano

Volcano is a System Batch-Job Scheduler made for High-Performance Workloads on Kubernetes. Volcano extends Kubernetes with functionalities that Kubernetes do not natively support. Some of these functionalities are critical when working with High-Performance Workloads, like “PodGroups”. In a scenario where a Framework might want to create multiple pods for its computation, the resources inside the cluster only allow for a few of them to be deployed. Applications could encounter deadlocks, requiring more pods to be deployed to progress. The Concept of PodGroups prevents Pods from being scheduled unless all of them can be scheduled.

The Volcano scheduler is based on the Kubernetes Scheduling Framework, influencing the scheduling cycle at the extension points. This way, Volcano can implement many scheduling policies, which High-Performance Batch Applications commonly use. Volcano is more concerned with policies around the actual scheduling algorithm. In contrast, the External-Scheduling-Interface, introduced in this work, focuses on aiding the development of the actual scheduling algorithm. In theory, algorithms could be implemented using Volcano, but since Volcano provides just a thin layer above the Kubernetes Scheduling Framework, using Volcano for the development of new scheduling algorithms does not seem like a plausible choice.

With both Frameworks supporting Batch Scheduling in different ways, the External-Scheduling-Interface and Volcano could complement each other, but there has been no further investigation.

6.2 Non Kubernetes

7

Conclusion and Future Work

7.1 Conclusion

7.2 Future Work

Bibliography

[1] github.com brancz/kube-rbac-proxy. https://github.com/brancz/kube-rbac-proxy. Accessed: 2022-03-09.

[2] GitHub.com gcp/spark-on-k8s-operator. https://github.com/GoogleCloudPlatform/spark-on-k8s-operator. Accessed: 2022-03-10.

[3] GitHub.com spotify/flink-on-k8s-operator. https://github.com/spotify/flink-on-k8s-operator. Accessed: 2022-03-10.

[4] kubernets.io concepts/pods. https://kubernetes.io/docs/concepts/workloads/pods/. Accessed: 2022-03-09.

[5] kubernets.io concepts/pods. https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/. Accessed: 2022-03-09.

[6] N. Capit, G. Da Costa, Y. Georgiou, G. Huard, C. Martin, G. Mounie, P. Neyron, and O. Richard. A batch scheduler with high level components. In *CCGrid 2005. IEEE International Symposium on Cluster Computing and the Grid, 2005.*, volume 2, pages 776–783 Vol. 2, 2005.

[7] Telmo da Silva Morais. Survey on frameworks for distributed computing: Hadoop, spark and storm. In *Proceedings of the 10th Doctoral Symposium in Informatics Engineering-DSIE*, volume 15, 2015.

[8] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for {Fine-Grained} resource sharing in the data center. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, 2011.

[9] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, New York, NY, USA, 2013. Association for Computing Machinery.

[10] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10)*, 2010.