Extending the Kubernetes scheduler for network resource awareness

Bachelor thesis by Thomas Kosiewski (Student ID: 2930013)
Date of submission: May 29, 2023

1. Reviewer: Prof. Dr. Felix Wolf
2. Reviewer: Dr. Aasem Ahmad
Darmstadt

Computer Science Department
Parallel Programming

Extending the Kubernetes scheduler for network resource awareness

Bachelor thesis by Thomas Kosiewski (Student ID: 2930013)

Date of submission: May 29, 2023

Darmstadt

Bitte zitieren Sie dieses Dokument als:
URN: urn:nbn:de:tuda-tuprints-263646
URL: http://tuprints.ulb.tu-darmstadt.de/26364
Jahr der Veröffentlichung auf TUprints: 2024

Dieses Dokument wird bereitgestellt von tuprints,
E-Publishing-Service der TU Darmstadt
http://tuprints.ulb.tu-darmstadt.de
tuprints@ulb.tu-darmstadt.de

Die Veröffentlichung steht unter folgender Creative Commons Lizenz:
Namensnennung 4.0 International
https://creativecommons.org/licenses/by/4.0/

This work is licensed under a Creative Commons License:
Attribution 4.0 International
https://creativecommons.org/licenses/by/4.0/

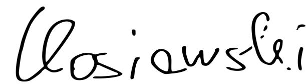
Erklärung zur Abschlussarbeit gemäß § 22 Abs. 7 APB TU Darmstadt

Hiermit erkläre ich, Thomas Kosiewski, dass ich die vorliegende Arbeit gemäß § 22 Abs. 7 APB der TU Darmstadt selbstständig, ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt habe. Ich habe mit Ausnahme der zitierten Literatur und anderer in der Arbeit genannter Quellen keine fremden Hilfsmittel benutzt. Die von mir bei der Anfertigung dieser wissenschaftlichen Arbeit wörtlich oder inhaltlich benutzte Literatur und alle anderen Quellen habe ich im Text deutlich gekennzeichnet und gesondert aufgeführt. Dies gilt auch für Quellen oder Hilfsmittel aus dem Internet.

Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Falle eines Plagiats (§ 38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.



Darmstadt, May 29, 2023

T. Kosiewski

Abstract

Kubernetes (K8s) has emerged as the de facto standard for distributed container workload orchestration in cloud and on-premises environments. Due to its open-source nature, strong separation of duty, and well-defined interfaces, Kubernetes creates abstraction layers between cluster operators, compute & storage providers, networking providers, and workloads authors. Developers can deploy their applications without needing thorough experience in the abovementioned fields to deploy and scale their applications. Instead, they can collaborate by utilizing existing resources and configurations and have their workloads run and distributed according to the specifications in the workload definitions.

In the current Kubernetes environment, scheduling decisions are based on CPU and memory requirements, neglecting other crucial resources such as network bandwidth. As the adoption of networking-intensive applications and workloads progresses, the need for network-aware scheduling becomes more pressing, as issues such as network congestion and overall system stability can degrade over time.

This thesis aims to improve the scheduler and ecosystem by incorporating plug-and-play extensions to the current scheduler and proposing a new scheduler that utilizes a different scheduling approach and incorporates algorithms optimization algorithms to find optimizations for the Kubernetes scheduling problems.

Experiments indicate that our solution outperforms the existing Kubernetes scheduler in solution quality and correctness, performing qualitatively higher resource distribution and guarantees. By deploying representative samples of network-demanding workloads in simulations, the extended scheduler ensured resource requirements for pods, while the default Kubernetes scheduler failed to do so. This improvement introduces a negligible computing cost to the scheduler. In addition, it avoids network congestion, idle cpu time, and overall higher resource usage on the machines, increasing network throughput and reducing application lags, avoiding slowdowns of two to three times the necessary time if networking resources were met.

Keywords: bin packing; CNCF; distributed algorithms; Kubernetes; scheduler; Quality of Service;

Goals and Objectives

This bachelor’s thesis aims to address the current Kubernetes scheduler limitations in regards to network bandwidth requirements and enable these as an additional scheduling attribute for Kubernetes workloads.

The goals are:

1. Study the existing literature related to optimization, scheduling problems and applicable scheduling algorithms.
2. Enable the Kubernetes scheduler to make decisions based on static bandwidth requirements in a standard Kubernetes cluster.
3. Propose a mechanic for performing dynamic scheduling, considering the actual usage of resources instead of static declarations.
4. Verify the proposed algorithms and extensions on benchmark instances and compare them with existing works.
5. Implement a simulation scenario using the kube-scheduler-simulator and KWOK to verify the proposed solution for correction and scalability.

The work will contribute to the growing body of research on resource-aware scheduling in the Kubernetes ecosystem, with the potential to improve the efficiency and performance of Kubernetes workloads.

Contents

1 Introduction **12**
- 1.1 Motivation . . . 12
- 1.2 Outline and Contributions . . . 12
- 1.3 Structure of the Thesis . . . 13

2 Overview **14**
- 2.1 Containers . . . 14
- 2.2 Kubernetes . . . 15
- 2.3 Current Kubernetes Scheduler . . . 17
- 2.4 Kubernetes Controllers . . . 18
- 2.5 Pod Disruption Budget . . . 20
- 2.6 Kubernetes Scheduling Framework and Extensibility . . . 20
- 2.7 Kubernetes Scheduler Simulator . . . 21

3 Related Work **23**
- 3.1 GPU resource scheduling . . . 23
- 3.2 Load-Aware Dynamic Scheduling . . . 23

4 Scheduling Approach **25**
- 4.1 Problem Statement . . . 25
- 4.2 Mathematical Formulation . . . 25
- 4.3 Network Extended Scheduler . . . 27
- 4.4 Novel adaptive scheduling approach . . . 28
  - 4.4.1 Illustration of the cyclic improvement process . . . 30

5 Algorithms integration into Kubernetes **34**
- 5.1 Integrating the Network Extended Scheduler . . . 34
- 5.2 Integrating Kube-scheduler-rs . . . 36
- 5.3 Benefits and Implications . . . 37

6 Simulation and Result Evaluation **38**
6.1 Benchmarks settings . . . 38
6.2 Simulation Scenarios . . . 40
6.2.1 Scenario 1: Fixed and strict network bandwidth requirements . . . 40
6.2.2 Scenario 2: Burstable network bandwidth requirements and node overcommitment . . . 41
6.2.3 Scenario 3: Workload distribution with mixed workloads . . . 44
6.2.4 Scenario 4: Workload distribution with mixed, burstable workloads 44
6.3 Scalability . . . 46
6.4 Resource utilization . . . 46
6.5 Application performance . . . 49

7 Conclusion **50**

List of Figures

2.1 Monolith vs Microservice Architectures 15
2.2 Kubernetes Cluster Architecture 16
2.3 Kubernetes scheduler extension points [11] 18
2.4 Architecture of a custom Kubernetes controller/operator 19
2.5 Kubernetes Scheduler Simulator showcasing the internal scoring and final scores of each plugin within the kube-scheduler 22

4.1 The proposed adaptive scheduling approach 29
4.2 Initial workload distribution with one pending pod 31
4.3 Second cycle, after bin packing the pending pod to a fitting node is yielding the lowest value. 32
4.4 Third cycle, after swapping a running pod from node A to B, utilizing tabu-list local search, yielding the lowest value. 33

6.1 CPU core usage during scheduling scenarios 47
6.2 Memory usage in MiB during scheduling scenarios 48

List of Tables

4.1 List of Symbols 27
4.2 Scoring terms 28
4.3 Normalization terms 28

List of Code Listings

5.1 Pod with CNI annotations 35

6.1 Kwok node definition 39
6.2 Pod with 200M total network bandwidth requirements scheduled by the default-scheduler 42
6.3 Pod with 200M total network bandwidth requirements scheduled by the network-extended-scheduler 43
6.4 Pod annotations for burstable network bandwidth 44
6.5 Deployment of CPU and memory-heavy application 45
6.6 Pod resource object for burstable CPU and memory 45

1 Introduction

This chapter introduces the problem of enhancing the Kubernetes scheduler with network resource awareness. It establishes the issue's importance, outline the thesis contribution and the subsequent chapters.

1.1 Motivation

The driving factor behind this thesis is the ever-increasing need for effective network resource management in Kubernetes environments. The emergence of identity-aware proxies, access control systems, and streaming applications such as Pomerium [15], Hashicorp's Boundary [2], StrongDM [17], and Teleport [17] has highlighted the importance of network bandwidth management for containerized applications. If network bandwidths and latencies cannot be guaranteed overall system stability will be impacted and can result in hard to recover error states in a given topology. This thesis aims to develop a solution that empowers the Kubernetes scheduler to allocate workloads based on their network bandwidth requirements, ensuring optimal resource distribution and improved overall performance.

1.2 Outline and Contributions

This section outlines the main objectives and contributions of the bachelor's thesis, which focuses on enhancing the scheduling mechanism in a container orchestration system, specifically Kubernetes. The research aims to achieve two primary goals:

- **Extending the native scheduler:** The thesis proposes developing a scheduler extension that enables the inclusion of network bandwidth requirements into the scheduling process. This extension provides an efficient means of incorporating this essential resource consideration by utilizing annotations on workloads. It enhances the existing bin-packing algorithm by filtering and scoring machines based on their static resource requirements.
- **Novel adaptive scheduling mechanism:** The thesis proposes a novel scheduling approach inspired by a reconciling controller. The proposed approach aims to reconcile the state of workloads and machines within the cluster, periodically evaluating and resolving inconsistencies. It responds to changes in workload resource requirements, the number of workloads, and the number of machines in the cluster, allowing for adaptive workload distribution and overcoming limitations of the current Kubernetes scheduling mechanism.

These contributions address critical challenges in container orchestration by improving network bandwidth management and dynamic workload distribution. In addition, they provide valuable insights and pave the way for advancements in scheduling mechanisms within container orchestration systems.

1.3 Structure of the Thesis

The thesis is structured as follows. Chapter 2 gives an overview to the relevant scheduling algorithms and Kubernetes related topics. Since the Kubernetes scheduler is distributing workloads to nodes already, chapter 3 presents the current approaches taken by both industry and academia to overcome its shortcomings and how those can relate to network bandwidth for workload scheduling in Kubernetes. The objective of chapter 4 is to propose algorithmic solutions regarding the shortcomings of the existing Kubernetes scheduler, whereas chapter 5 will showcase the integration of the algorithm into Kubernetes. Chapter 6 provides an evaluation in regards to the quality of the solution, the resources necessary to achieve it, and the time it took. Finally, chapter 7 provides a conclusion to the thesis and potential future works that can be conducted.

2 Overview

The primary goal of this overview is to familiarize the reader with the fundamental concepts required for comprehending the thesis. This chapter's content is concise and targeted, avoiding excessive details unrelated to the central thesis objectives. We will examine the interaction among various components and their significance in expanding the Kubernetes scheduler's network resource awareness.

2.1 Containers

Parallel programming often divides workloads into smaller tasks that can be executed concurrently on multiple processors. Similarly, modern software applications can be separated into smaller, isolated units called containers, which can operate independently and communicate with one another. Consequently, containers offer an efficient method for packaging, deploying, and managing applications, enabling developers to concentrate on code development without concerning themselves with the underlying infrastructure.

As shown in figure 2.1 the container and microservice approach enables the creation of loosely coupled services interacting with each other, without a central point of failure or centralized bottleneck. Additionally, this kind of architecture has gained a lot of popularity in recent years, due to it's ability to scale horizontally, as more instances of a micro service can be launched automatically, and due to the ability to separate services by team and domain, instead of having all of them in one central, monolith service.

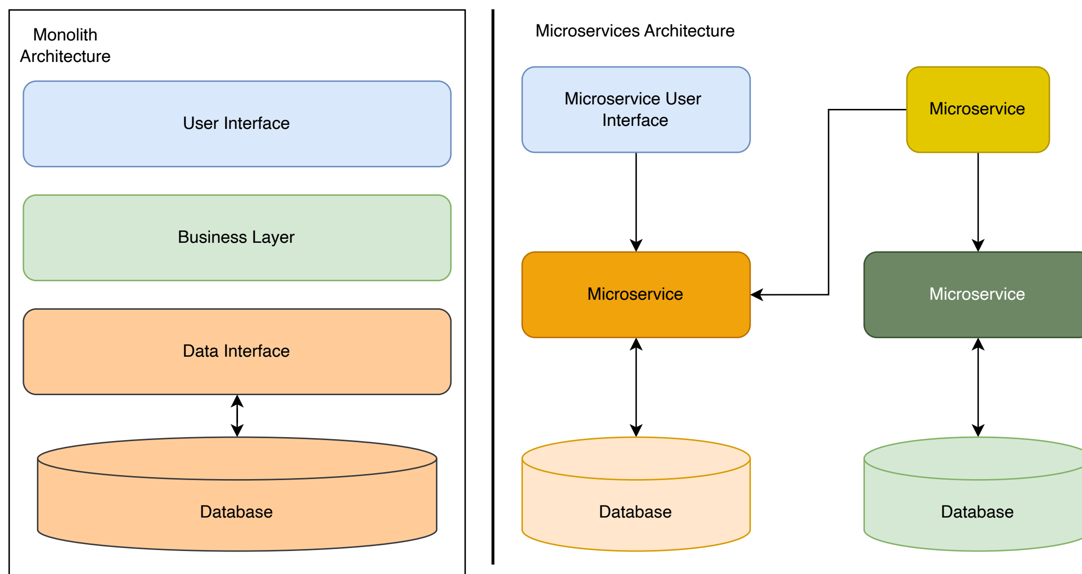


Figure 2.1: Monolith vs Microservice Architectures

2.2 Kubernetes

Kubernetes is an open-source platform that automates containerized applications' deployment, scaling, and management. It organizes containers into logical groupings called pods and distributes them across several machines referred to as nodes, ensuring that applications operate efficiently and maintain high availability.

Originating at Google [1], Kubernetes was inspired by the company's internal container orchestration system, Borg [18] , which was created to manage the vast scale of Google's infrastructure. Kubernetes was open-sourced in 2014 and has since become the de facto standard for container orchestration.

Kubernetes itself is an assembly of multiple critical services that work in concert to sustain the cluster's desired state:

- **Nodes** can be physical or virtual machines that execute containerized applications. They may serve as worker nodes, which host user-defined workloads, or as control plane nodes, containing the control plane components that govern the entire cluster.

- **Pods** represent the most basic deployable units within the Kubernetes ecosystem. Pods can incorporate one or more side-car containers that utilize shared network and storage resources. Additionally, each pod is allocated a unique IP address within the cluster and can expose several ports for communication.
- The **Control Plane** includes the API server, etcd datastore, controller manager, and scheduler to manage the overall state of the cluster. These components work together to ensure that the cluster maintains its desired configuration. In order to upkeep the cluster's stability and functionality, the control plane components perform tasks such as resource allocation and workload distribution, including system monitoring.

Figure 2.2 outlines roughly the interplay between the Kubernetes components running on nodes and the control plane.

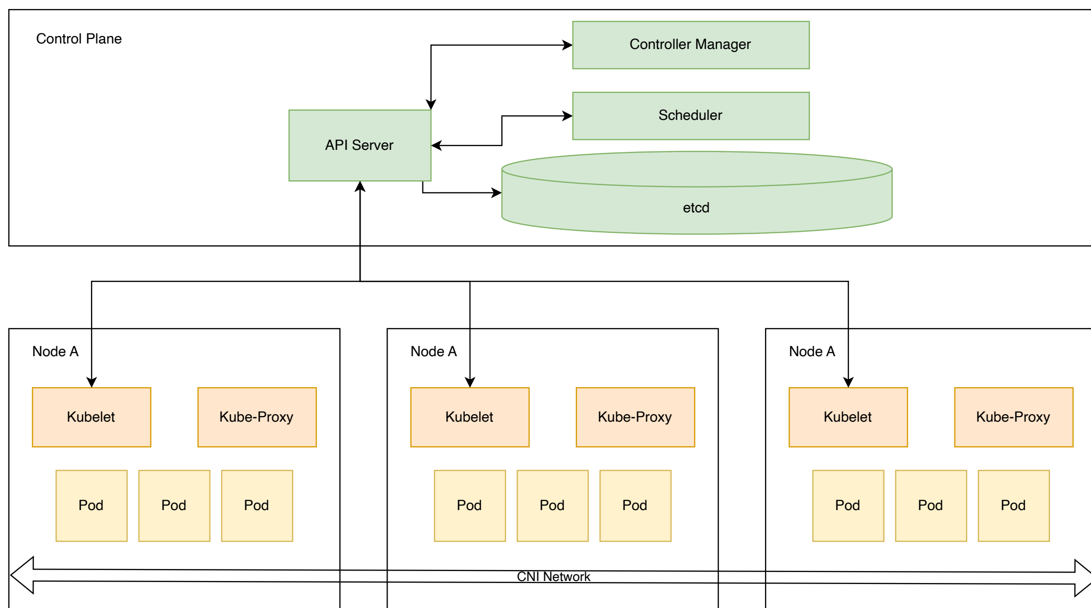


Figure 2.2: Kubernetes Cluster Architecture

2.3 Current Kubernetes Scheduler

Kubernetes employs the scheduler (kube-scheduler) [10] to determine which nodes in the cluster should run newly created pods. The scheduling problem is a bin-packing problem with NP-Hard complexity. The nodes can be considered as the ”bins”, pods can be regarded as the ”objects”, and the volumes of both can be represented by the total resource availabilities (CPU, memory, etc.) a node has and total resource requirements that a workload demands.

Scheduling pods on Kubernetes is a two-part operation: filtering and scoring nodes. During the filtering phase, the scheduler filters out nodes that cannot cater to the pod’s requirements. These might be due to resource limitations, taints, or tolerations. The scoring phase then ensues, where the remaining nodes are scored based on established rules. The node with the highest score is considered the ideal host for the pod. This approach is generally viewed as a heuristic algorithm, meaning it employs a set of rules for decision-making rather than a complete exploration of all possible options. In the scoring phase, Kubernetes uses multi-criteria decision-making, simultaneously considering various factors, such as resource availability and affinity rules. One could compare the scheduling strategy of the Kubernetes scheduler to a ”best bin” methodology. The ”first bin” strategy would imply placing a pod on the first node capable of accommodating it without regard to other considerations. However, the Kubernetes scheduler evaluates several factors to score each node and chooses the best one to run the pod.

This scheduler provides a framework for extension development, which includes disk and data localities, inter-pod affinities and anti-affinities, node name filtering, node ports, resource suitability, topology spreads, and the availability of CPU and memory. These add-ons, portrayed by the green arrows in Figure 2.3, enrich the scheduling process by delivering specialized function implementations.

The Kubernetes scheduler operates through two separate cycles, as Figure 2.3 illustrates scheduling and binding. The scheduler initiates with PreEnqueue operations before entering the scheduling cycle, subsequently sorting the pods to be scheduled. Scheduler extensions can also augment the sorting mechanism. Following the sorting stage, optional pre-filter functions provided by the extensions, along with their filter functions, are applied to exclude unfeasible nodes further.

Post-filtering, the scheduler administers the pre-scoring functions, followed by scoring functions, culminating in normalizing those values. The scheduler applies user-defined weighting to specific extension scores during the normalization process. Consequently, one

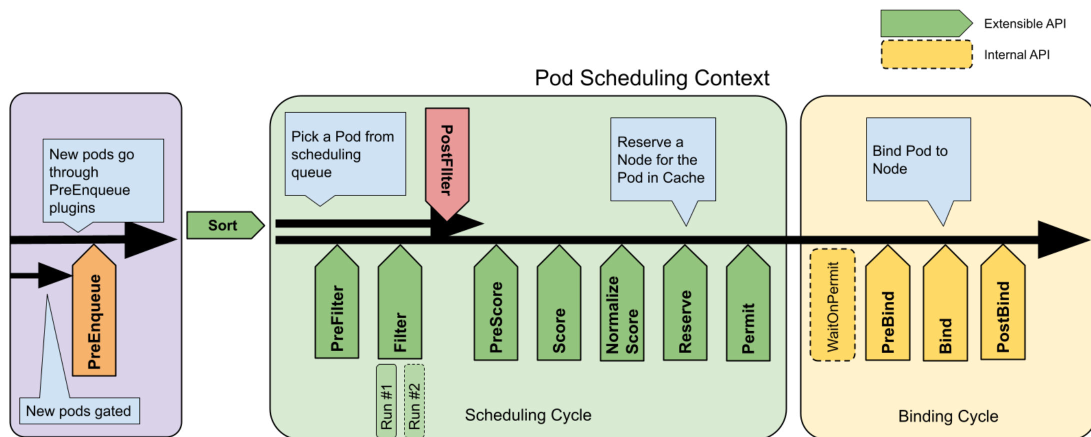


Figure 2.3: Kubernetes scheduler extension points [11]

node will emerge with the highest score, reserving the pod's resources in the scheduler cache and approving binding to that node.

During the binding phase, the scheduler invokes a binding API call that binds the pod to the node on which the pod shall run.

However, the Kubernetes scheduler does not consider network resources of nodes and pod bandwidth requirements and can thus not factor in network bandwidth requirements during resource allocation and scheduling. Integrating network resource consideration into the Kubernetes scheduler is crucial for optimizing resource allocation and averting performance deterioration due to network bottlenecks. By allowing the scheduler to weigh network bandwidth requirements alongside CPU and memory resources in scheduling decisions, network resource consideration helps prevent situations in which workloads with high network demands run on nodes lacking sufficient network capacity and, in turn, enhances overall application performance and resource utilization.

2.4 Kubernetes Controllers

Kubernetes controllers [6] are vital in maintaining the cluster's desired state by continuously reconciling the observed state with the intended state. These controllers are

engineered to monitor cluster resource changes and take appropriate action to achieve the desired state.

A controller operates in a loop that involves the following steps:

1. Observe the current state of the cluster.
2. Compare the observed state with the desired state.
3. Execute necessary actions to reconcile any discrepancies between the two states.

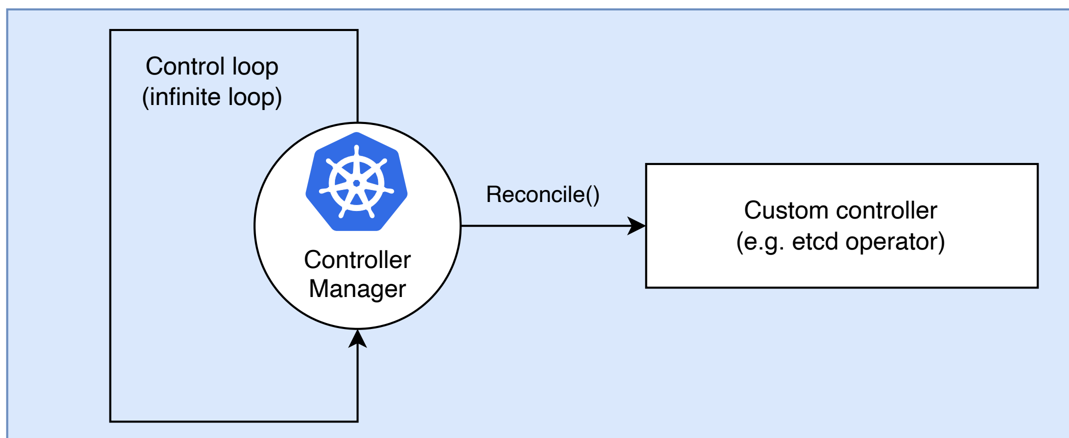


Figure 2.4: Architecture of a custom Kubernetes controller/operator

The reconciling nature of controllers provides several benefits in terms of stability and reliability:

- **Self-healing**: Controllers perpetually oversee the cluster and autonomously rectify any inconsistencies, ensuring the preservation of the desired state in case of failures or alterations.
- **Scalability**: The reconciling loop enables controllers to adjust to cluster changes, such as scaling up or down based on resource requirements, without human intervention.
- **Extensibility**: Kubernetes offers a flexible framework for creating custom controllers tailored to specific use cases or requirements, fostering the development of domain-specific controllers that align with application needs.

By harnessing the inherent stability and reliability of Kubernetes controllers, the proposed network-aware scheduler can effectively manage network resources and guarantee optimal performance for containerized applications.

2.5 Pod Disruption Budget

A Pod Disruption Budget (PDB) [16] is a Kubernetes feature that enables users to specify the minimum number of available replicas for a particular application, ensuring that the application remains accessible during voluntary disruptions, such as maintenance or updates.

2.6 Kubernetes Scheduling Framework and Extensibility

The Kubernetes scheduling framework [9] allows for the development of custom scheduler plugins that can be integrated into the existing scheduler. These plugins can introduce new scheduling features or modify the scheduler's behavior to consider additional scheduling attributes, such as network bandwidth requirements. In addition, there are several methods to extend the Kubernetes scheduler:

1. **Custom Scheduler Plugins**: Developers can create custom scheduler plugins using the Kubernetes scheduling framework. These plugins can be written in Go and integrated directly into the scheduler codebase. [9]

2. **Scheduler Extenders**: The Kubernetes scheduler can call these external services during its scheduling process. Scheduler extenders can be written in any language and communicate with the scheduler via RESTful APIs. [8]

3. **Multiple Schedulers**: Multiple Schedulers: Kubernetes also supports running multiple scheduler instances, each with its configuration and behavior. This approach allows different workloads to be scheduled using distinct scheduling algorithms or policies, depending on their specific requirements. [5]

By leveraging these extensibility methods, developers can create a network-aware Kubernetes scheduler that considers network bandwidth requirements when making scheduling decisions. This enhanced scheduler can optimize resource allocation, prevent network bottlenecks, and improve overall application performance.

2.7 Kubernetes Scheduler Simulator

Verifying the efficiency of the suggested network-aware Kubernetes scheduler is vital for confirming that it satisfies the intended goals and functions optimally across diverse scenarios. Therefore, a simulation-oriented method will be adopted, using the Kubernetes Scheduler Simulator [12] established by the Kubernetes Special Interest Group (SIG).

The Kubernetes Scheduler Simulator is a practical resource explicitly crafted to assist developers in examining and understanding the behavior of Kubernetes schedulers. It enables users to emulate various scheduling situations, scrutinize the scheduler's choices, and assess its comprehensive performance, as can be seen in detail in Figure 2.5.

By implementing the Kubernetes Scheduler Simulator, developers can glean invaluable knowledge about the network-sensitive scheduler's performance under various conditions, such as varying network requirements, resource constraints, and workload allocations. This information can be harnessed to adjust and optimize the scheduler's behavior, ensuring it affords the most advantageous resource allocation and workload distribution for network-demanding applications.

Figure 2.5: Kubernetes Scheduler Simulator showcasing the internal scoring and final scores of each plugin within the kube-scheduler

3 Related Work

Many research projects and initiatives have delved into expanding the capabilities of the Kubernetes scheduler to address limitations concerning resources beyond just CPU and memory.

3.1 GPU resource scheduling

In scheduling deep learning tasks on Kubernetes, KubFBS [14] and GPUShare [4] are examples of scheduler extensions to support additional resource types in Kubernetes. KubFBS [14] is a scheduling system that has been developed by academia with the express goal of increasing efficiency when scheduling deep learning tasks on Kubernetes. Conversely, GPUShare is a scheduler extender developed by industry professionals, namely Alibaba Cloud, which significantly enhances the Kubernetes scheduler's capacity to support GPU resource allocations.

GPUShare and KubFBS utilize the same proposed method, namely an extension to the Kubernetes scheduler, to solve a different yet related problem to extended resource scheduling in Kubernetes.

3.2 Load-Aware Dynamic Scheduling

For addressing the dynamic, load-aware scheduling in Kubernetes, the crane-scheduler [3] proposes a solution for monitoring the actual resource usage of workloads and adjusting resource allocation based on real-time resource demands. The crane-scheduler project demonstrates the potential benefits of incorporating dynamic resource allocation based on

actual resource usage, which could be adapted to address network bandwidth requirements in our problem statement.

The implementation differs in that it considers the current resource load onto the nodes but does not re-evaluate the current distribution or perform rescheduling. Thus it carries the same shortcomings as the current Kubernetes scheduler but can initially distribute workloads more efficiently.

4 Scheduling Approach

The primary aim of this chapter is to present a scholarly and abstract problem statement focused on enhancing the Kubernetes scheduler by incorporating network resource awareness.

4.1 Problem Statement

Within the Kubernetes ecosystem, the Kubernetes scheduler makes scheduling decisions that consider resource demands, node restrictions, and affinity rules. Nonetheless, the default Kubernetes scheduler mainly concentrates on CPU and memory resources, frequently neglecting other vital resources, such as network bandwidth.

One can express the issue this thesis intends to tackle as follows:

> Given a Kubernetes cluster comprising multiple nodes with varying network bandwidth capabilities and workloads with diverse network bandwidth demands, how can we enhance the Kubernetes scheduler to effectively allocate network resources, avoid network congestion, and optimize overall application performance?

4.2 Mathematical Formulation

Let N be the set of nodes in the Kubernetes cluster and W be the set of workloads to be scheduled. Then, for each node $i \in N$, let B_i denote the available network bandwidth capacity, and for each workload $j \in W$, let b_j represent the required network bandwidth. The goal is to assign each workload $j \in W$ to node $i \in N$, minimizing the network

congestion and balancing the overall application performance while considering the CPU and memory resource constraints. We introduce a binary decision variable x_{ij}:

$$x_{ij} = \begin{cases} 1 & \text{if workload } j \text{ is assigned to node } i \\ 0 & \text{otherwise.} \end{cases}$$

Our objective is to minimize network congestion while allocating network resources effectively:

$$\text{minimize: } \sum_{i \in N} \sum_{j \in W} (B_i - b_j * x_{ij})^2$$

according to the following constraints:

- Workload assignment constraint: Each workload must be assigned to exactly one node. $\sum_{i \in N} x_{ij} = 1, \forall j \in W$
- Network bandwidth capacity constraint: The sum of the required network bandwidths of the workloads assigned to a node must not exceed the node's network bandwidth capacity. $\sum_{j \in W} b_j * x_{ij} \leq B_i, \forall i \in N$
- CPU and memory resource constraints: Let C_i and M_i denote the available CPU and memory capacities of node i, and let c_j and m_j represent the required CPU and memory resources of workload j. The sum of the required CPU and memory resources of the workloads assigned to a node must not exceed the node's CPU and memory capacities.

$$\text{CPU capacities: } \sum_{j \in W} c_j * x_{ij} \leq C_i, \forall i \in N$$

$$\text{Memory capacities: } \sum_{j \in W} m_j * x_{ij} \leq M_i, \forall i \in N$$

- Binary decision variable constraint: $x_{ij} \in \{0, 1\}, \forall i \in N, \forall j \in W$

The resulting mathematical formulation is a Quadratic Mixed-Integer Program (QMIP) that minimizes network congestion while effectively allocating network resources and considering CPU and memory constraints.

N	Set of nodes in the Kubernetes cluster
W	Set of workloads to be scheduled
i	A node in the node set N
B_i	Available network bandwidth capacity for node i
j	A workload in the workload set W
b_j	Required network bandwidth for workload j
x_{ij}	Binary decision variable indicating node assignment
C_i	Available CPU capacity for node i
M_i	Available memory capacity for node i
c_j	Required CPU capacity for workload j
m_j	Required memory capacity for workload j

Table 4.1: List of Symbols

4.3 Network Extended Scheduler

The primary intent of this algorithm, subsequently called the Network Extended Scheduler, abbreviated to NES, is to build upon the existing Kubernetes native solution, extending its current capabilities to tackle bandwidth requirements. In addition, by modifying the inherent bin-packing algorithm of Kubernetes, we improve its functionality in terms of network demands. This modification to the native bin-packing algorithm is discussed throughout this section, focusing on how bandwidth consideration impacts different stages like filtering, scoring, and others.

As presented in the introduction section 2.3, the standard Kubernetes scheduler employs a heuristic best-fit bin-packing algorithm, providing a framework for extension development.

NES implements the Filter, Score, and NormalizeScore functions made available by this extension framework.

- The **Filter** function interprets and parses a pod's network bandwidth requirements as defined in its annotations. Subsequently, nodes unable to meet these networking needs are filtered out, ensuring only capable nodes will be used further.
- The **Score** function comes into action by deducting the pod's necessary network bandwidth of the node's available network resources, providing the difference as score. Specifically:

Let $B = \{b_1, b_2, b_3, ..., b_n\}$ denote all node bandwidths.
Let bw denote a workload's bandwidth requirements.

Table 4.2: Scoring terms

Then each node's score s_i is calculated as $s_i = b_i - bw$, i.e., the node's bandwidth is subtracted from the required bandwidth, resulting in a node's individual score.

- The **NormalizeScore** function adjusts the previously obtained score of each node to be within the range of 0 to 100. For example:

Let $S = \{s_1, s_2, s_3, ..., s_n\}$ denote all node scores.
Let $s_{min} = \min(S)$ denote the minimum of all scores, so that $\forall s \in S, s \geq s_{min}$.
Let $s_{max} = \max(S)$ denote the maximum of all scores, so that $\forall s \in S, s \leq s_{max}$.
Let s_{node} be a score such that $s_{min} \leq s_{node} \leq s_{max}$ denote a node score to be scaled.
We require that $s_{min} < s_{max}$.

Table 4.3: Normalization terms

Then:

$$s_{node} \rightarrow \frac{s_{node} - s_{min}}{s_{max} - s_{min}} * 100$$

will scale s_{node} linearly into $[0, 100]$.

4.4 Novel adaptive scheduling approach

The proposed adaptive scheduling approach, called kube-scheduler-rs, abbreviated to KSRS, periodically reconciles the state of the world. During each run, the scheduler enters a scheduling cycle. It executes multiple parallel algorithms, yielding a proposed workload distribution called an intermediate state in Figure 4.1.

This parallel computation is then invoked N times with the newly discovered and improved workload distribution unless none of the algorithms find a better workload distribution, in which case the loop exits early. Once the scheduling cycle yields an improved workload

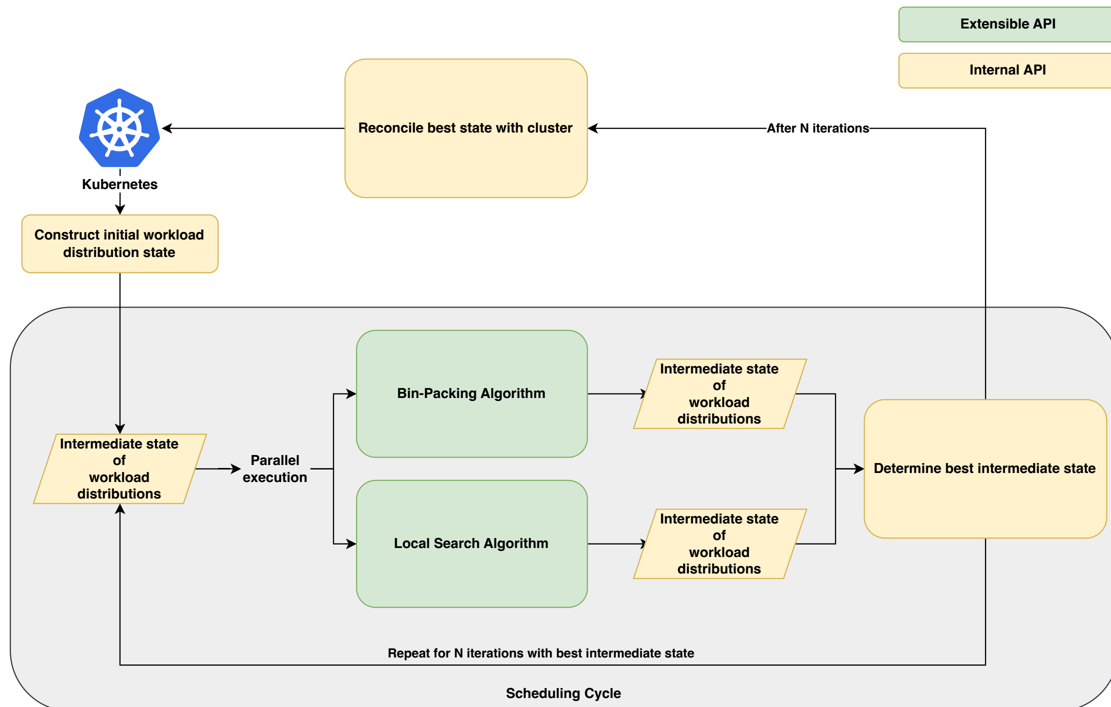


Figure 4.1: The proposed adaptive scheduling approach

distribution, the reconcile function applies the desired workload distributions within the Kubernetes cluster, and the run is complete.

In this thesis, we propose a scheduling approach that uses a bin-packing algorithm, like the one found in Kubernetes, referenced in Section 4.3, combined with a tabu-search-based local search algorithm. This way, we ensure that the distribution of tasks across multiple runs will not underperform compared to the default Kubernetes scheduler on a per-pod basis.

The local search process depicted in Figure 4.1 employs a tabu search-based algorithm. In this procedure, pods - individual and in groups, scheduled or pending - are swapped, with the tabu list logging these swaps, and an evaluation function evaluates the subsequent distributions. We aim to sidestep potential pitfalls, such as being caught in poorly scoring regions or plateaus, and instead perform searches into various parts of the search space.

When both algorithms have reached a suggested workload distribution, KSRS evaluates

these distributions based on a set of heuristic rules and either reintroduces those proposed distributions back into the scheduling cycle, encouraging a cycle of distribution revision and further optimization or applies the distribution to the Kubernetes cluster, once a higher scoring distribution was found.

4.4.1 Illustration of the cyclic improvement process

The following is an illustration of the cyclic improvement process, assuming there is only one resource and each node has a capacity of 100 units of said resource. Pod A requires 33 units and Pod B requires 50 units to run.

In Figure 4.2, the initial utilization of Node A is 100, Node B 50, and Node C 0. Conclusively, the amount of unused resources is 0 for Node A, 50 for Node B and 100 for Node C. According to the formula defined in 4.2, this would yield a score of $12500 = 0^2 + 50^2 + 100^2$.

In the next cycle, scheduling the pending pod to Node C utilizing bin-packing lowers the score the most, as the utilization of Node C increases to 50, as seen in Figure 4.3. The score is now $5000 = 0^2 + 50^2 + 50^2$.

Subsequently, utilizing a tabu-list local search in the following cycle, Figure 4.4, identifies an even superior workload placement for another pod. Moving one pod from Node A to Node B further decreases the score to 3878, as $33^2 + 17^2 + 50^2$ is an even lower score than found in the last cycle. Conclusively, this method allows KSRS to consider multiple pods for preemption and reassignment.

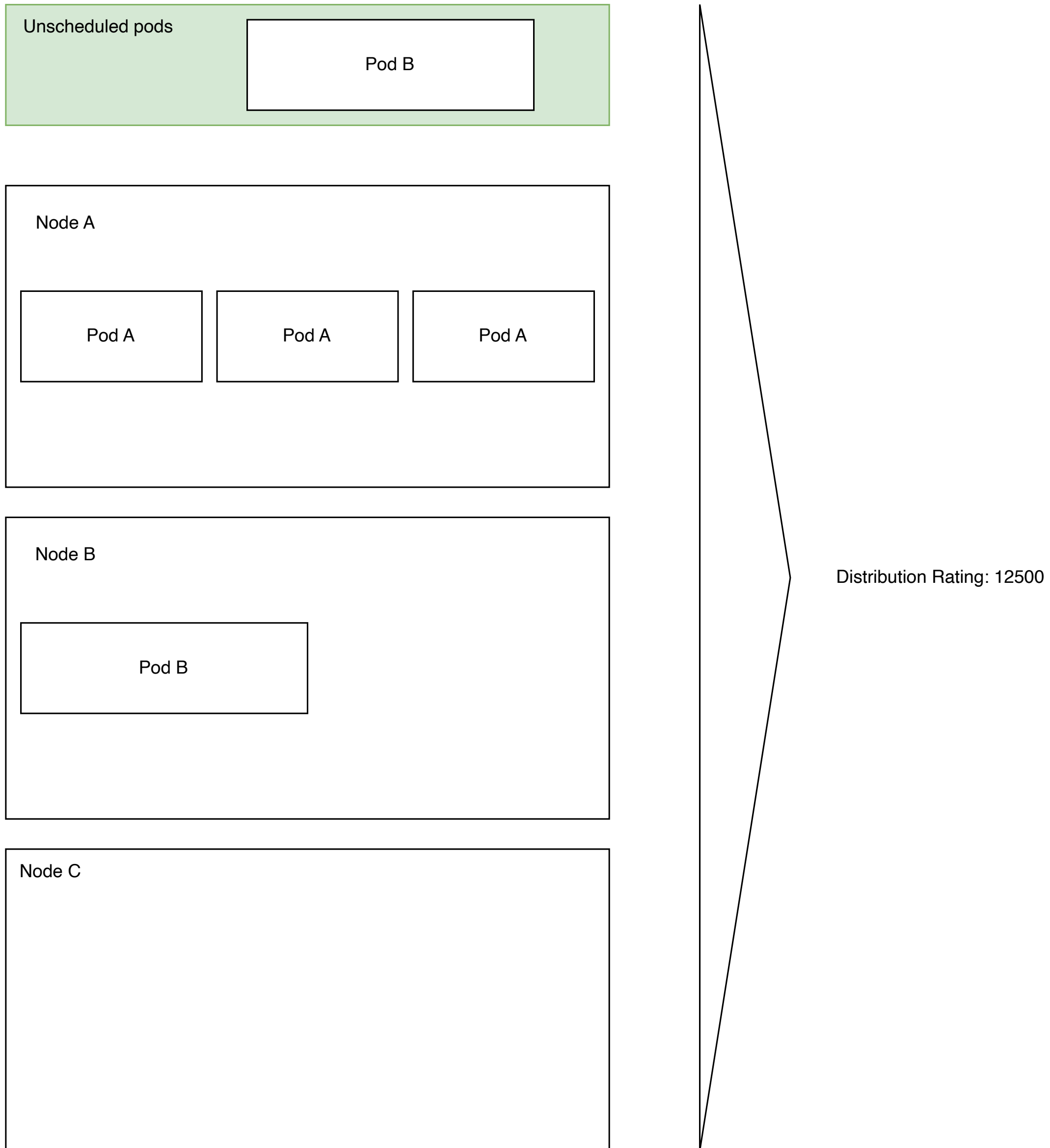


Figure 4.2: Initial workload distribution with one pending pod

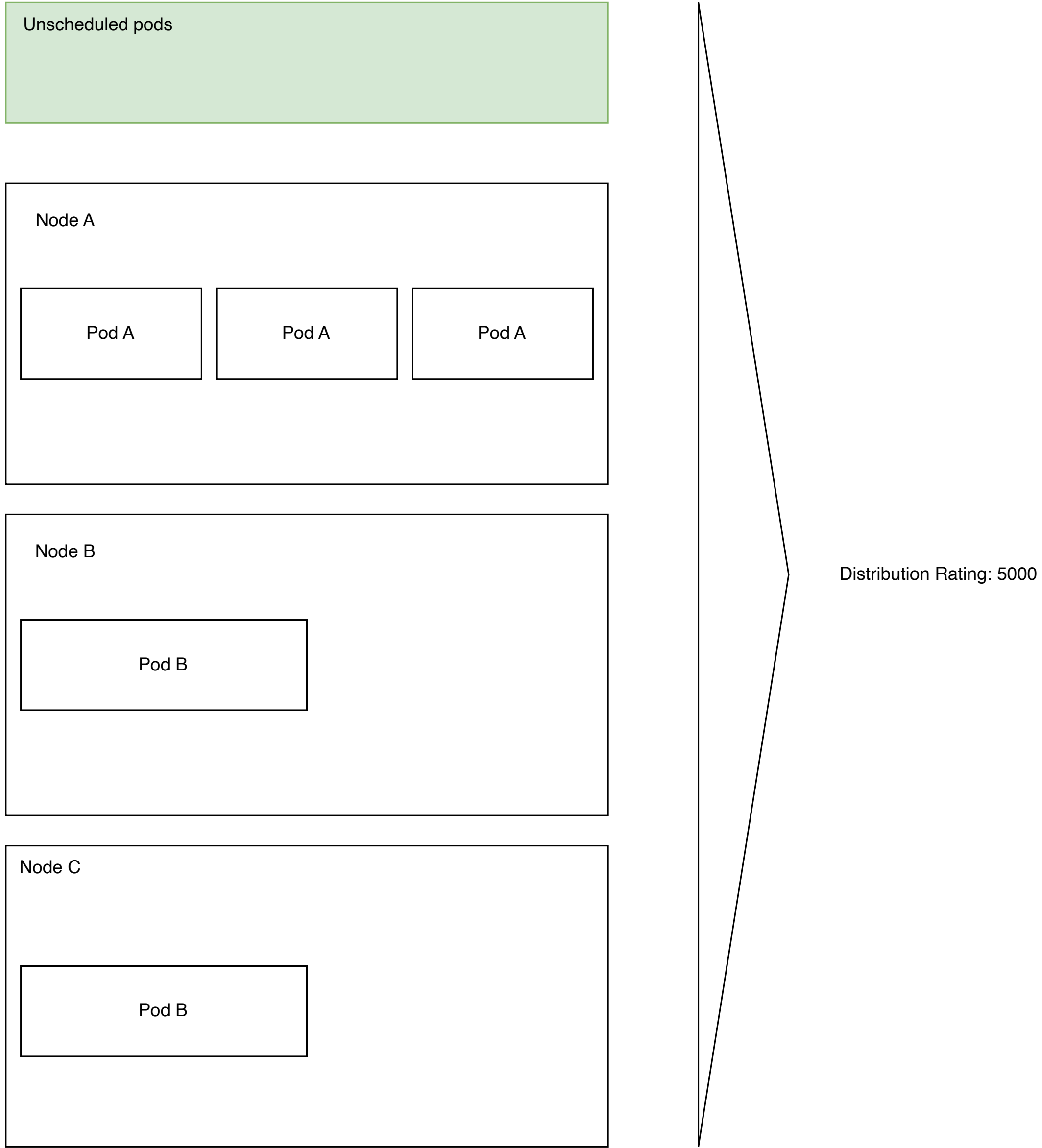


Figure 4.3: Second cycle, after bin packing the pending pod to a fitting node is yielding the lowest value.

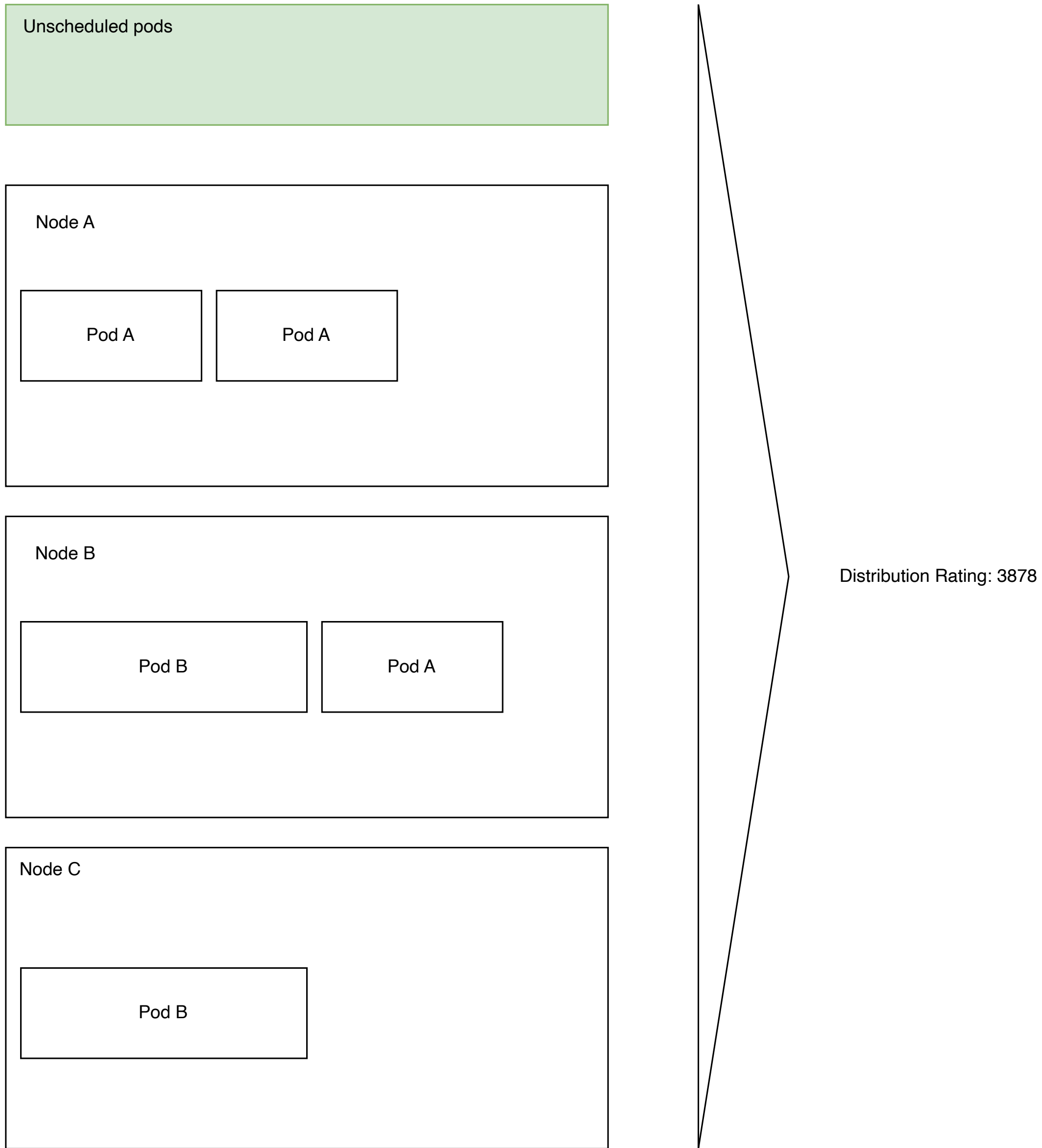


Figure 4.4: Third cycle, after swapping a running pod from node A to B, utilizing tabu-list local search, yielding the lowest value.

5 Algorithms integration into Kubernetes

In this chapter, we discuss the methods employed to tackle the deficiencies in the Kubernetes scheduler concerning network bandwidth allocation. Our strategy consists of augmenting the Kubernetes scheduler using a custom extension and proposing an independent, tailored scheduler that uses various scheduling algorithms to enhance workload distribution.

5.1 Integrating the Network Extended Scheduler

To overcome the constraints of the standard Kubernetes scheduler, we have implemented and integrated NES into Kubernetes and broadened the bin-packing algorithm by considering network bandwidth requirements. Moreover, by integrating network bandwidth annotations transmitted to the Container Network Interface (CNI), the plugin introduces a new scheduling aspect to the Kubernetes scheduler. In Code Listing 5.1, one can see the pod object response the Kubernetes API server sends to the Kubernetes scheduler, which the scheduler then passes to all scheduler extensions as arguments. In addition, lines 8 and 10 include the relevant bandwidth information for the CNI to introduce bandwidth limits, and NES now utilizes that information.

NES assesses only static network bandwidth requirements, guaranteeing that pods with substantial network needs are placed on nodes with adequate network resources. The placing happens by filtering out nodes that do not have the available resources to run the pod and then scoring the remaining nodes based on the total and normalized amount of available network bandwidth and other resources, with configurable weights applied to each normalized score.

There are multiple ways in which one can integrate NES:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
  namespace: kube-scheduler-rs
  annotations:
    # Limits the ingress bandwidth to 2Mbit/s
    networking.k8s.io/ingress-bandwidth: 2M
    # Limits the egress bandwidth to 2Mbit/s
    networking.k8s.io/egress-bandwidth: 2M
spec:
  containers:
    - name: my-container
      image: registry.k8s.io/pause:2.0
      resources:
        requests:
          cpu: 2
          memory: 100Mi
        limits:
          cpu: 4
          memory: 1Gi
```

Code Listing 5.1: Pod with CNI annotations

- One could merge the extension's source code into the source of the Kubernetes scheduler. This way, one could use the "out of the box" extension without further maintenance on the cluster itself. However, merging back into the Kubernetes source is lengthy and can take months or years to merge and be released.
- A proposed alternative is building a custom image and either swapping out the default Kubernetes scheduler with one's custom-built image or running the two schedulers in parallel.

By NES only introducing a new plugin to the scheduler, one can non-intrusively prevent network congestion without significant changes in the code or introducing further components into the Kubernetes cluster. As a result, overall application performance and system stability can improve but might result in suboptimal overall resource allocation.

5.2 Integrating Kube-scheduler-rs

Alongside the NES, we are proposing the design of a novel adaptive scheduling approach, KSRS, that employs a generational method to address the scheduling problem, drawing inspiration from genetic algorithms and their fitness functions from the Machine Learning field. This novel scheduler merges multiple scheduling algorithms, such as bin packing and tabu-list local search, to identify the most efficient workload distribution within a specified time frame. This combined approach enables KSRS to utilize the strengths of each algorithm to find the most effective workload distribution within a given time frame.

The bin-packing algorithm seeks to optimize resource utilization by packing workloads as tightly as possible, reducing wasted resources, and maximizing available capacity. Conversely, the tabu-list local search algorithm explores the scheduling solution space by iteratively making minor adjustments to the existing solution and avoiding previously encountered solutions, aiding it in escaping local optima and discovering improved solutions over time.

By integrating these algorithms in a generational fashion, KSRS can benefit from the bin-packing algorithm's effectiveness in initial workload placement and the tabu-list local search algorithm's capacity to explore and refine solutions iteratively. For instance, it might initially schedule a pod using bin-packing and subsequently utilize tabu-list local search in the next scheduling cycle to identify an even superior placement for another pod. Additionally, this method allows KSRS to consider multiple pods for preemption and reassignment, creating space for new pods while adhering to pod disruption budgets.

By considering PDBs when making scheduling decisions, KSRS guarantees that the rescheduling and reassignment of pods do not breach the stated availability requirements, minimizing the potential impact on application performance and reliability. [7]

KSRS's structure resembles a standard Kubernetes controller more than the existing scheduler implementation. This reconciling framework simplifies the adaptation of the scheduler to work with dynamic values. It can be repurposed for schedulers that prefer to use dynamic or current usage values instead of static ones.

Integrating this scheduler into an existing Kubernetes cluster requires the cluster administrator to create a new deployment and specify its name in the Kubernetes pod resources. Subsequently, the default scheduler will not schedule this pod, and the proposed scheduler can schedule it as described above.

This method allows running both scheduler deployments in parallel, reducing risks and allowing fine-grained scheduling per namespace and workload demand.

5.3 Benefits and Implications

Implementing NES and KSRS addresses significant limitations concerning the network bandwidth allocation of the default Kubernetes scheduler. By considering network bandwidth requirements in conjunction with CPU and memory constraints, these solutions enable more informed scheduling decisions that avoid network bottlenecks and optimize overall application performance.

The KSRS's capability to reschedule running workloads offers a distinct advantage. It allows for ongoing workload distribution and resource allocation optimization in response to evolving application demands and network conditions. In this manner, a dynamic scheduler can be implemented, as with each reconciliation run, previous decisions can be revised, and more efficient distributions applied to the cluster. This architecture also goes hand in hand with the introduction of ”In-Place Update of Pod Resources” in the Kubernetes 1.27 release.

6 Simulation and Result Evaluation

In this chapter, we examine the NES implementation, created to mitigate the inadequacies of the Kubernetes scheduler concerning network bandwidth allocation. Additionally, we will propose a strategy for conducting benchmarks on the resulting solutions to evaluate their performance and efficacy in optimizing resource allocation within Kubernetes clusters.

6.1 Benchmarks settings

To verify the correctness and quality of the resulting workload distribution by the default Kubernetes scheduler and the NES implementation, one will create diverse Kubernetes cluster using KWOK [13], outlined in detail in each subsequent scenario, to compare the resulting workload distributions. Each node in the cluster will have a 'node.kubernetes.io/network-limit: 1Gi' annotation, indicating a 1 Gbit/s network bandwidth limit, 32 CPU cores and 256Gi of memory, see Code Listing 6.1.

All scenarios will run on a 2023 16-inch MacBook Pro with an Apple M2 Pro chipset and 32 GB of memory. The docker desktop VM hosting the KWOK-managed Kubernetes cluster will have 6 CPU cores and 8GB of memory allocated.

Section 5.1 outlined running the scheduler with the custom extension as a secondary scheduler. By defining the "schedulerName" field in the pod's definition, one can explicitly define which scheduler will be used to schedule the pod.

Each scenario will have individually adjusted workload definitions, that will be comparable to the Code Listings in 6.2 and 6.3, yet with adequate changes for the given simulation setting.

We track the scheduler's resource utilization using a custom-built Go command line application that connects directly to the system's docker daemon and efficiently collects

```yaml
apiVersion: v1
kind: Node
metadata:
  annotations:
    node.alpha.kubernetes.io/ttl: "0"
    node.kubernetes.io/network-limit: "1Gi"
  labels:
    beta.kubernetes.io/arch: arm64
    beta.kubernetes.io/os: linux
    kubernetes.io/arch: arm64
    kubernetes.io/hostname: kwok-node-0
    kubernetes.io/os: linux
    kubernetes.io/role: agent
  name: kwok-node-0
status:
  allocatable:
    cpu: "32"
    memory: 256Gi
    pods: "110"
  capacity:
    cpu: "32"
    memory: 256Gi
    pods: "110"
  nodeInfo:
    architecture: arm64
    bootID: f9d3750e-8d48-4e73-867a-b144a06e17f8
    containerRuntimeVersion: containerd://1.6.19-k3s1
    kernelVersion: 5.15.49-linuxkit
    kubeProxyVersion: v1.26.4+k3s1
    kubeletVersion: v1.26.4+k3s1
    machineID: ""
    operatingSystem: linux
    osImage: K3s dev
    systemUUID: ""
  phase: Running
```

Code Listing 6.1: Kwok node definition

CPU and memory usage over time. The captured output is then analyzed and visualized in a Jupyter Lab Notebook, utilizing Pandas and Matplotlib.

6.2 Simulation Scenarios

In order to evaluate the performance and effectiveness of both the NES and default Scheduler implementations, we suggest carrying out a series of benchmarks within simulated Kubernetes environments. The benchmarks will concentrate on these key performance indicators:

1. **Scalability**: Investigate the schedulers' capacity to manage increasing numbers of nodes, pods, and workloads and ascertain the time for the schedulers to make and implement scheduling decisions.
2. **Resource utilization**: Evaluate CPU, memory, and network bandwidth utilization efficiency.
3. **Application performance**: Examine the influence of the schedulers on the performance of containerized applications operating within the Kubernetes clusters.

The benchmarking strategy will entail establishing multiple Kubernetes clusters with diverse workloads and network demands. The custom scheduler plugin and the generational custom scheduler will manage these clusters, and their performance will be observed and compared to that of the default Kubernetes scheduler.

6.2.1 Scenario 1: Fixed and strict network bandwidth requirements

In this scenario, we assume that the workloads need their requested resource requirements to be guaranteed by the scheduler. Examples of these workload classes include live video transcoding, media transcoding, and file transfer.

One will be creating a cluster consisting out of 200 nodes by adjusting the node's name in Code Listing 6.1 line 14, and applying the definition to the KWOK-managed cluster.

Utilizing the workload definitions shown in Code Listing 6.2 and 6.3, we deploy pods with equal network bandwidth requests and limits into dedicated testing namespaces. Thus the Kubernetes quality of service class for these pods will be ”Guaranteed”. One utilizes

the pod definitions shown in Code Listing 6.2 for the default Kubernetes scheduler and Code Listing 6.3 for the NES implementation. The only differences for the pod definition between the two listings is line 26 explicitly defining the scheduler to be used, and the namespace in line 12.

Each namespace will hold 1500 pods, requiring a total of 300Gi. However, only 200Gi are available in the cluster. Subsequently, the scheduler should only schedule 1000 of those 1500 pods. Thus, the remaining 500 pods shall remain pending indefinitely, as one cannot provide the workloads with their requested network bandwidths. Unless the cluster administrator scales the cluster horizontally, i.e., adding new nodes to the cluster, adding new resources availabilities to it, or once running workloads are scaled down or complete their task and free up their allocated resources to be used by other pods. This blocking behavior is the desired behavior in Kubernetes, as, by definition, resource requests must be matched for a workload to start.

6.2.2 Scenario 2: Burstable network bandwidth requirements and node overcommitment

Unlike the first scenario, we assume the pod's workload requirements are burstable, meaning pods will have a base network bandwidth guaranteed, e.g., an average minimum to work correctly and a much higher limit on the network bandwidth limit. As in scenario one, we will create a KWOK-managed cluster of 200 nodes by adjusting the node's name in Code Listing 6.1 line 14 and applying the definition to the KWOK-managed cluster.

In order to implement this scenario, one adjusts the network bandwidth annotation in lines 21 to 24 in Code Listings 6.2 and 6.3 to the annotations shown in Code Listing 6.4.

As in scenario one, the deployments will consist of 1500 pods, with each deployment residing in a dedicated namespace explicitly specifying the scheduler used for scheduling. Each namespace will hold 1500 pods, requiring 150Gi of network bandwidth, with a limit of 300Gi of combined ingress and egress bandwidth. The scheduler should schedule all 1500 pods, evenly balancing the workloads with regards to network bandwidth requirements.

```yaml
apiVersion: v1
kind: Namespace
metadata:
  labels:
    kubernetes.io/metadata.name: stress-test-default-scheduler-1500-100
  name: stress-test-default-scheduler-1500-100
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: stress-test
  namespace: stress-test-default-scheduler-1500-100
spec:
  replicas: 1500
  selector:
    matchLabels:
      app: pause
  template:
    metadata:
      annotations:
        kubernetes.io/ingress-bandwidth: 100M
        kubernetes.io/egress-bandwidth: 100M
        kubernetes.io/ingress-request: 100M
        kubernetes.io/egress-request: 100M
      labels:
        app: pause
    spec:
      schedulerName: default-scheduler
      containers:
        - name: pause-container
          image: gcr.io/google_containers/pause:3.2
          resources:
            requests:
              cpu: "0.1"
              memory: "10Mi"
            limits:
              cpu: "0.1"
              memory: "10Mi"
```

Code Listing 6.2: Pod with 200M total network bandwidth requirements scheduled by the default-scheduler

```yaml
apiVersion: v1
kind: Namespace
metadata:
  labels:
    kubernetes.io/metadata.name: stress-test-network-extended-scheduler-1500-100
  name: stress-test-network-extended-scheduler-1500-100
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: stress-test
  namespace: stress-test-network-extended-scheduler-1500-100
spec:
  replicas: 1500
  selector:
    matchLabels:
      app: pause
  template:
    metadata:
      annotations:
        kubernetes.io/ingress-bandwidth: 100M
        kubernetes.io/egress-bandwidth: 100M
        kubernetes.io/ingress-request: 100M
        kubernetes.io/egress-request: 100M
      labels:
        app: pause
    spec:
      schedulerName: network-extended-scheduler
      containers:
        - name: pause-container
          image: gcr.io/google_containers/pause:3.2
          resources:
            requests:
              cpu: "0.1"
              memory: "10Mi"
            limits:
              cpu: "0.1"
              memory: "10Mi"
```

Code Listing 6.3: Pod with 200M total network bandwidth requirements scheduled by the network-extended-scheduler

```
kubernetes.io/ingress-bandwidth: 100M
kubernetes.io/egress-bandwidth: 100M
kubernetes.io/ingress-request: 50M
kubernetes.io/egress-request: 50M
```

Code Listing 6.4: Pod annotations for burstable network bandwidth

6.2.3 Scenario 3: Workload distribution with mixed workloads

This scenario analyzes the scheduling behavior with mixed types of workload requirements. We will deploy two distinct deployments, one primarily requiring network bandwidth and the other workload only requiring CPU and memory while not having any networking requirements, into a newly created cluster to the likes of the cluster in scenario one. We will use Code Listings 6.2 and 6.3 for this, and for the CPU and memory-intensive workload, we will use Code Listing 6.5.

As in the first, fixed bandwidth scenario, we will guarantee pods the availability of their resource requirements by applying unmodified Code Listings 6.2 and 6.3. First, the CPU and memory-demanding Kubernetes deployment will be applied, followed by the network-demanding deployment.

This scenario showcases the balanced distribution of networking bandwidth across diverse workloads in a cluster.

6.2.4 Scenario 4: Workload distribution with mixed, burstable workloads

This fourth scenario builds on the third scenario but introduces the option for both deployments to be burstable in resource requirements, thus overcommitting node resources. The networking burst ability is introduced by replacing lines 21 to 24 in Code Listings 6.2 and 6.3 with the annotations showcased in Code Listing 6.4. In contrast, one introduces workload burst ability by replacing lines 19 to 25 in Code Listing 6.5 with the resources object showcased in Code Listing 6.6.

This scenario aims to showcase the balanced distribution of diverse workloads in a cluster across overcommitted node resources and the resulting distribution of workloads.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: stress-test-cpu-memory-workload
spec:
  replicas: 1600
  selector:
    matchLabels:
      app: cpu-memory
  template:
    metadata:
      labels:
        app: cpu-memory
    spec:
      schedulerName: network-extended-scheduler
      containers:
        - name: pause-container
          image: gcr.io/google_containers/pause:3.2
          resources:
            requests:
              cpu: "2"
              memory: "16Gi"
            limits:
              cpu: "2"
              memory: "16Gi"
```

Code Listing 6.5: Deployment of CPU and memory-heavy application

```yaml
resources:
    requests:
        cpu: "1"
        memory: "8Gi"
    limits:
        cpu: "2"
        memory: "16Gi"
```

Code Listing 6.6: Pod resource object for burstable CPU and memory

6.3 Scalability

While performing scalability testing on KWOK-managed Kubernetes clusters, both NES and the default scheduler could handle the increasing number of nodes and pods to distribute.

As seen in the Figures in 6.1, the total time for scheduling each scenario is comparable for both schedulers, which traces back to API server rate limiting and local caching. The schedulers consistently took less than 2 seconds to schedule a pod to a node in all four simulation scenarios with clusters consisting of 200 nodes. This delay mainly traces back to the Kubernetes API server imposing rate limiting and, thus, the scheduler keeping a cache, which is periodically fetched and updated.

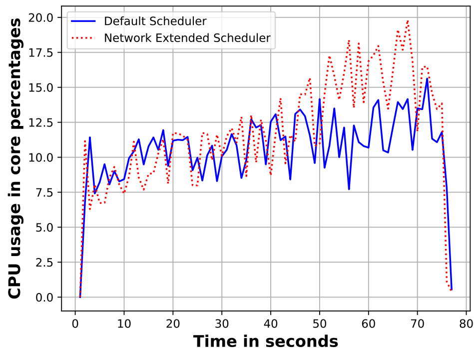
Hence, from a scalability perspective, both schedulers make and implement scheduling decisions in the same timeframe without lagging behind the other, and both schedulers are not putting additional pressure on the Kubernetes API server, even with an increasing number of nodes and pods to schedule.

6.4 Resource utilization

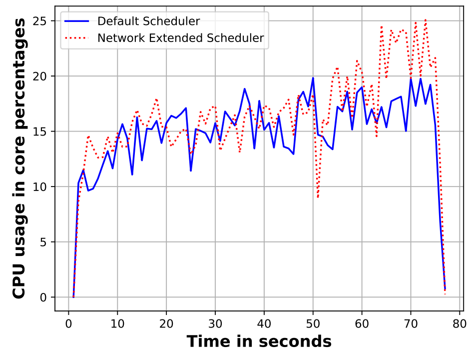
Regarding CPU usage, the charts in Figure 6.1 showcase that NES occasionally has a slightly higher CPU usage than the default scheduler. This additional CPU usage can be attributed to the calculations performed by the network scheduling extension or measurement and rounding errors.

On the other hand, the charts in Figure 6.2 portray a comparable memory utilization across both schedulers, that like the CPU usage, are occasionally higher and occasionally lower but require the same memory to operate.

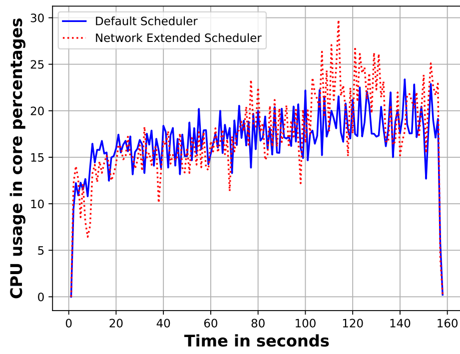
As outlined in the Scalability section, both schedulers keep an internal cache due to Kubernetes API server rate limiting and optimization purposes. Thus the network bandwidth utilization of NES remains equal to the utilization of the default scheduler, as the cache invalidation and refreshing is a separate component of the scheduler that has not been modified.



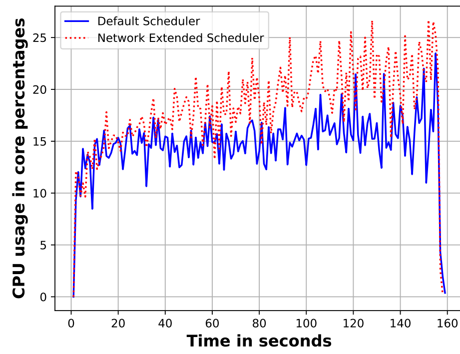
(a) Scenario 1



(b) Scenario 2

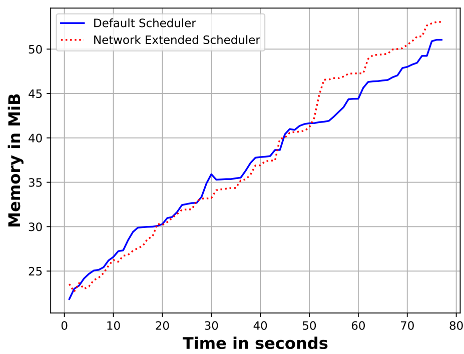


(c) Scenario 3

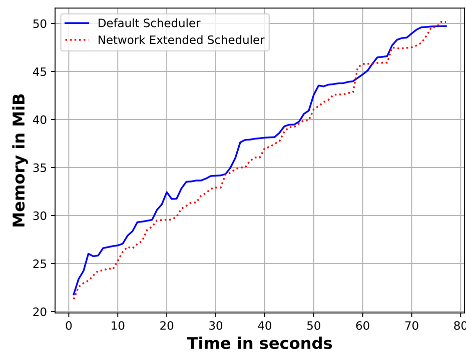


(d) Scenario 4

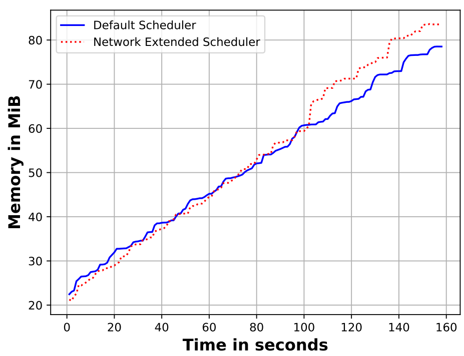
Figure 6.1: CPU core usage during scheduling scenarios



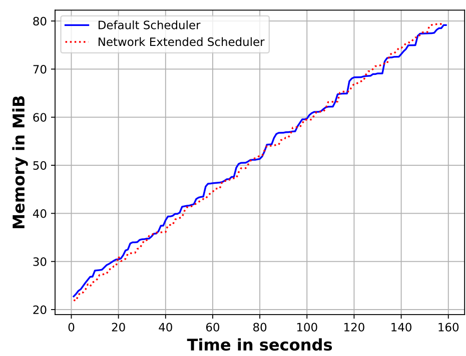
(a) Scenario 1



(b) Scenario 2



(c) Scenario 3



(d) Scenario 4

Figure 6.2: Memory usage in MiB during scheduling scenarios

6.5 Application performance

In contrast to both previous sections, the pod distributions resulting from NES differ enormously from the ones resulting from the default scheduler.

Within the first and third test scenarios, the scheduler provided pods with their required constant network bandwidth, meaning that applications could perform at 100In the case of video transcoding and file transfer services, these workloads did not experience any slowdowns or increased pings due to network congestion. In contrast, the default scheduler only balanced out pods based on the number of pods on a node, disregarding networking needs. Unfortunately, this disregard leads to data transfers slowing down by 50-75

On the other hand, in scenarios two and three, NES distributed pods with network requirements evenly across nodes in the entire cluster, balancing the workloads and demands. The workloads had a predictable behavior with higher usage, linearly increasing in network congestion, as even with overcommitted nodes, the percentage of the overcommitment is fixed.

In contrast, using the default scheduler resulted in imbalanced distributions of network-demanding workloads. As a result, the default scheduler heavily overcommitted some nodes while underutilizing others from a networking bandwidth perspective. This imbalance impacted the workloads, as in their burstable nature, the performance became unpredictable and, with higher usage, led to higher pings and delayed operations more than necessary.

7 Conclusion

During this bachelor's thesis, the driving goal was to resolve a pressing issue – the lack of network-bandwidth awareness in Kubernetes scheduling. The vision was to extend the Kubernetes scheduler's capabilities and propose a new, more dynamic Kubernetes scheduler capable of making real-time decisions based on static bandwidth requirements and accounting for actual resource usage.

The journey has seen the creation of a scheduler extension and a design proposal for a novel adaptive scheduling approach for Kubernetes. The extension, crafted using the Kubernetes extension framework, enabled the scheduler to read network bandwidth requirements directly from pod annotations, enriching the existing bin-packing algorithm by filtering and scoring nodes based on these specifications. This advancement allows cluster operators to better integrate network bandwidth resource requirements into the standard Kubernetes scheduler, facilitating a more efficient scheduling process.

However, proposing the design of a novel adaptive scheduling approach for a Kubernetes scheduler is an area for further research. This scheduler design, mirroring the concept of Kubernetes controllers, undertakes a periodic reconciliation of the state of pods and nodes within the cluster. This constant updating means it can adapt swiftly to changes in workload resource requirements, the number of pods and workloads, or even the number of nodes within the cluster. Furthermore, its ability to continuously reassess and adapt to external changes ensures optimal workload distribution, a capability that the existing Kubernetes scheduler needs to improve.

The robustness of these improvements was confirmed through testing using the kube-scheduler-simulator and KWOK. In addition, they demonstrated the practicality of the proposed solutions and pointed the way to future enhancements in the Kubernetes scheduling ecosystem.

In conclusion, this thesis has significantly advanced the discourse on network-bandwidth awareness in Kubernetes scheduling. It has not only met the set objectives but also made a

meaningful contribution to the broader discipline of resource-aware scheduling in Kubernetes. Furthermore, the work carried out here lays the groundwork for future research in this area, pointing towards a future with even more refined resource management within Kubernetes environments, such as introducing adaptive scheduling mechanisms and ways of implementing those in the future.

Bibliography

[1] *About Google*. URL: https://about.google (visited on 03/31/2023).

[2] *Boundary*. HashiCorp Inc. URL: https://www.boundaryproject.io/ (visited on 03/31/2023).

[3] *Crane-scheduler*. URL: https://github.com/gocrane/crane-scheduler (visited on 03/31/2023).

[4] *GPU Sharing Scheduler Extender in Kubernetes*. Alibaba Cloud. URL: https://github.com/AliyunContainerService/gpushare-scheduler-extender (visited on 03/31/2023).

[5] *Kubernetes – Configure Multiple Schedulers*. The Linux Foundation. URL: https://kubernetes.io/docs/tasks/extend-kubernetes/configure-multiple-schedulers/ (visited on 03/31/2023).

[6] *Kubernetes – Controllers*. The Linux Foundation. URL: https://kubernetes.io/docs/concepts/architecture/controller/ (visited on 03/31/2023).

[7] *Kubernetes – Disruptions*. The Linux Foundation. URL: https://kubernetes.io/docs/concepts/workloads/pods/disruptions/ (visited on 03/31/2023).

[8] *Kubernetes – Scheduler Extender*. The Linux Foundation. URL: https://github.com/kubernetes/design-proposals-archive/blob/main/scheduling/scheduler_extender.md (visited on 03/31/2023).

[9] *Kubernetes – Scheduling Framework*. The Linux Foundation. URL: https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/ (visited on 03/31/2023).

[10] *Kubernetes Scheduler*. The Linux Foundation. URL: https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/ (visited on 03/31/2023).

[11] *Kubernetes scheduler extension points*. URL: https://d33wubrfki0l68.cloudfront.net/a2935907046b492bc7ee8b9a10c54fffd6b5a4a0/834e0/images/docs/scheduling-framework-extensions.png (visited on 05/19/2023).

[12] *Kubernetes scheduler simulator*. The Linux Foundation. URL: https://github.com/kubernetes-sigs/kube-scheduler-simulator (visited on 03/31/2023).

[13] *KWOK (Kubernetes WithOut Kubelet)*. URL: http://kwok.sigs.k8s.io (visited on 05/10/2023).

[14] Zijie Liu et al. "KubFBS: A fine-grained and balance-aware scheduling system for deep learning tasks based on kubernetes". In: *Concurrency and Computation: Practice and Experience* 34.11 (2022), e6836. DOI: https://doi.org/10.1002/cpe.6836. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.6836. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.6836.

[15] *Pomerium*. Pomerium Inc. URL: https://pomerium.com/ (visited on 03/31/2023).

[16] *Specifying a Disruption Budget for your Application*. The Linux Foundation. URL: https://kubernetes.io/docs/tasks/run-application/configure-pdb/ (visited on 03/31/2023).

[17] *StrongDM*. StrongDM, Inc. URL: https://www.strongdm.com/ (visited on 03/31/2023).

[18] Abhishek Verma et al. "Large-scale cluster management at Google with Borg". In: *Proceedings of the European Conference on Computer Systems (EuroSys)*. Bordeaux, France, 2015.