

Improving Cluster Scheduling Resiliency to Network Faults

by

Sara Qunaibi

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2023

© Sara Qunaibi 2023

Author's Declaration

This thesis consists of material all of which I authored or co-authored: please see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

The results of this work are included in two papers. The first and main paper has been submitted to The 42nd International Symposium on Reliable Distributed Systems (SRDS 2023). I am the first author of this paper and it has been done in collaboration with Sreeharsha Udayashankar under the supervision of our advisor Samer Al-Kiswany. Sreeharsha provided technical support to help resolve failure or performance issues in my evaluation and provided feedback on the TAS scheduler design.

The second paper has been published at the ACM Transactions on Computer Systems (TOCS) titled "Partial Network Partitioning". In this paper I contributed the discussion around design pitfalls of modern scheduling techniques in data analytics engines.

List of papers:

1. Sara Qunaibi, Sreeharsha Udayashankar, Samer Al-Kiswany, TAS: Topology-Aware Scheduling, Symposium on Reliable Distributed Systems (Under Review), 2023.
2. Basil Alkhatib, Sreeharsha Udayashankar, Sara Qunaibi, Ahmed Al-Quraan, Mohammad Alfatafta, Wael Al-Manasrah, Alex Depoutovitch, Samer Al-Kiswany, Partial Network Partitioning, ACM Transactions on Computer Systems, 2022.

In my MMath study, I contributed to the following publications that are not presented in my thesis:

1. Asim Waheed, Sara Qunaibi, Diogo Barradas, Zachary Weinberg, Darwin's Theory of Censorship: Analysing the Evolution of Censored Topics with Dynamic Topic Models, Workshop on Privacy in the Electronic Society, 2022
2. Huiyi Ning, Sreeharsha Udayashankar, Sara Qunaibi, Karl Knopf, Xi He, Benchmarking Differentially Private Graph Algorithms, Workshop Theory and Practice of Differential Privacy, 2021

Abstract

We present a comprehensive empirical study of the impact partial network partitions have on cluster managers in data analysis frameworks. Our study shows that modern scheduling approaches are vulnerable to partial network partitions. Partial partitions can lead to a complete cluster pause or a significant loss of performance.

To overcome the shortcoming of the state-of-the-art schedulers, we design the topology-aware scheduler (TAS). TAS incorporates the current network connectivity information when making a scheduling decision, to allocate fully connected nodes for a given application. TAS effectively hides partial partitions from applications. Our evaluation of a TAS prototype shows that it can tolerate partial network partitions, eliminate application halting or significant loss of performance.

Acknowledgements

I would like to start by thanking Professor Samer Al-Kiswany without whom I would not have had the pleasure of experiencing such a fruitful and enlightening time during my masters.

I want to thank my parents; Rana and Sadeq, Thank you for being my support system, my motivation and my inspiration. Everything I am today is because of you and for that I will always be grateful.

I want to thank my fiance, Ibrahim. Thank you for being my home. For providing me with the support that I needed, I wouldn't have been able to complete this without you.

My siblings, my best friends, AlHamed, Maisan, Shaheen and Abdulwadood, thank you for being you.

And finally, I would like to thank my fellow WASL members for the all fun times we've had, in particular I'd like to thank Harsha, for being patient with me from the very beginning and Hala, for being my bestest friend abroad.

Table of Contents

Author’s Declaration	ii
Statement of Contributions	iii
Abstract	iv
Acknowledgements	v
List of Figures	viii
List of Tables	ix
1 Introduction	1
2 Background	3
2.1 Network Partitioning	3
2.2 Scheduling in Data Analytics Systems	5
2.2.1 Mesos	6
2.2.2 Kubernetes	7
3 The Impact of Partial Network Partitions on Scheduling	9
3.1 Methodology	9
3.2 Spark Standalone Cluster Manager	11

3.3	Mesos	12
3.4	Kubernetes	14
3.5	Summary	17
4	Topology Aware Scheduling	19
4.1	Connectivity Monitoring Overlay	20
4.2	Scheduler Metadata Service	21
4.3	Scheduler Design	22
5	Implementation	25
6	Evaluation	26
6.1	Fault Tolerance	26
6.2	Overhead Evaluation	27
6.3	Performance Under Partial Partitions	28
7	Related Work	33
8	Conclusion and Future Work	36
	References	37

List of Figures

2.1	Partial Network Partition	4
2.2	Apache Spark	6
2.3	K8s Deployment	8
3.1	Spark Standalone	11
3.2	Mesos as a CM.	13
3.3	K8s as a CM	15
4.1	TAS Design.	20
4.2	Connectivity Matrix of a 4 node cluster.	21
6.1	Benchmarking TAS and Kubernetes baseline scheduler against an increasing workload. The standard deviation for these experiments is less than 6%.	27
6.2	Average execution time of word count application under W-W partition. K8s, TAS, and TAS PNP have a standard deviation less than 3%. The figure shows the standard deviation for K8s PNP.	28
6.3	Average execution time of word count application under D-W partition. K8s, TAS, and TAS PNP have a standard deviation less than 3%. The figure shows the standard deviation for K8s PNP.	30
6.4	Average execution time of word count application under M-W partition. K8s, TAS, and TAS PNP have a standard deviation less than 5%. The figure shows the standard deviation for K8s PNP.	31
6.5	Average execution time of word count application under M-D partition.	31

List of Tables

3.1	Summary of the impact of partial partitions. H indicates that the application stops making progress until the partial partition is fixed. P indicates that partial partition causes performance degradation. - indicates that there is no impact on the running application. R indicates one repeat of task execution. We discuss TAS in Section 6.	17
-----	---	----

Chapter 1

Introduction

Modern large-scale applications in domains such as data analysis [69, 21], stream processing [28, 43], and online services [20], use hundreds of machines that are often dispersed across multiple data centers. These services are expected to provide high availability, performance, and resource utilization, despite failures in networks, devices, or software components.

Recent studies [12, 9, 11] show that network partitions, and partial network partitions in particular, pose a special challenge for system design. Partial partitions are a network partition that divides the network into three groups (say G1, G2, and G3) such that two groups (say G1 and G2) are disconnected, while G3 is able to communicate with all nodes in the cluster. Studies of failure reports [9] in production-quality systems show that this network fault often leads to catastrophic failures and is often due to design faults.

In my thesis, we focus on studying the impact of partial network partitions on large scale platforms. In these platforms, scheduling is a core technique for improving system reliability, performance, and resource efficiency. We conduct our study using state-of-the-art schedulers, when used for scheduling data analytics applications. Nevertheless, our insights and proposed solution are applicable beyond data analytics applications.

We start by conducting a comprehensive empirical study of the impact of partial partitions on modern schedulers. Our study includes Kubernetes [47], Mesos [37], and Spark native scheduler [69]. We identify the stages in an application execution that can be impacted by a network partition then use fault injection to empirically evaluate the impact of a partition at these stages. Our study shows that modern schedulers are vulnerable to partial partitions. In the majority of cases, partial partitions caused complete application pause until the partition is fixed, or caused significant performance degradation, up to $11\times$ longer execution time.

By studying the design of state-of-the-art schedulers, we identify a key shortcoming. Modern schedulers assume that network connectivity is transitive i.e. if the scheduler can reach node A, and A can reach node B, then the scheduler can reach node B. The scheduler also assumes that if it can reach node A, all other nodes can reach A and vice-versa. These fundamental assumptions are violated under partial network partitions, leading to failures with costly consequences.

To overcome the shortcomings of the state-of-the-art scheduler, we built the Topology-Aware Scheduler (TAS). TAS uses cluster connectivity information to inform its scheduling decisions. When a partial network partition occurs, TAS identifies the nodes that are still connected, and allocates fully connected nodes to new applications. If a partial partition happens during an application execution, TAS allocates the affected nodes to new ones to guarantee that each application is placed on a subset of nodes that are fully connected.

We implemented TAS by extending the Kubernetes scheduler. Our evaluation with standard data analytics workloads show that TAS eliminates all the negative impacts of partial partitioning. With TAS, applications never get stuck, and their performance is not degraded significantly. Compared to state-of-the-art schedulers TAS improves the application runtime by up to $7\times$.

The rest of this thesis is organized as follows. We present an overview of the design of modern schedulers in chapter 2, our study methodology in chapter 3, and the results of our study in chapter 3. We present TAS design in chapter 4 and our evaluation in chapter 6. We conclude in chapter 8.

Chapter 2

Background

In this chapter we present background related to network partition faults and scheduling techniques in Spark.

2.1 Network Partitioning

The scale and complexity of networked systems has grown exponentially, significantly increasing the impact of system failures regardless of their magnitude.

A network partition is a fault that causes nodes to divide into groups that are unable to communicate with one another. These network failures can have a significant impact on service availability and performance [60, 12]. Network Partitioning faults are common; Google reports experiencing 40 network partitions in two years [34], Microsoft reports that 70% of the downtime is caused by network partitions [33], and Turner et. al found that network partitions occur almost once every 4 days in the California-wide CENIC network [66].

Network partitions have a wide array of causes. They can manifest in geo-replicated systems because of a loss of connectivity between data centers [66], while switch failures can cause network partitions within a data center [33]. Network partitions caused by correlated failures are common [38, 33, 66], often caused by system-wide maintenance tasks [34, 33]. On a single node, NIC [54] or software failures can partition a node that may host multiple VMs.

Network partitions are amongst the most complex failures as they cannot be masked by the network layer and impact system semantics. Taking network partitions into account

complicates system design, fault analysis, and debugging. This is largely due to the limitations presented by the CAP theorem [32] which states that any distributed data store can only provide two of the three following guarantees: Consistency, Availability, and Partition Tolerance.

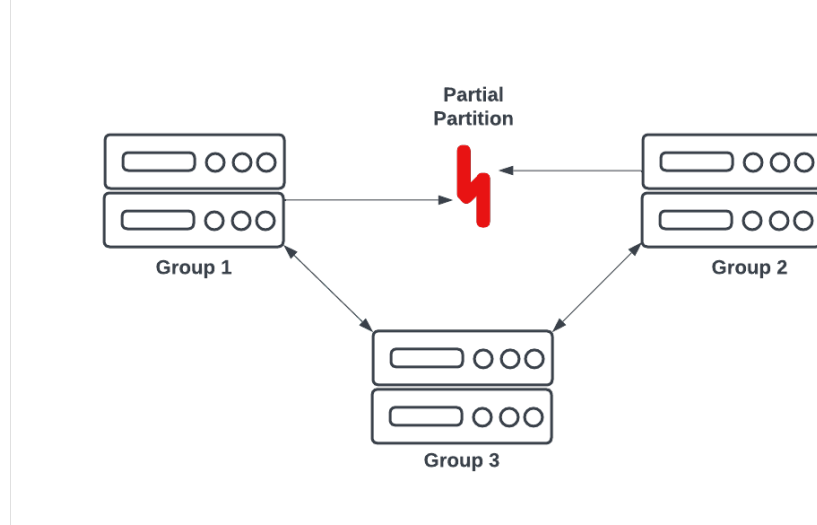


Figure 2.1: Partial Network Partition

Partial Network Partitioning. The most complex network partitions are partial network partitions. Partial network partitions disrupt the communication among some but not all nodes within a cluster. Figure 2.1 demonstrates how a partial network partition can divide a cluster into three separate groups. The nodes in groups 1 and 2 are not able to communicate with one another while nodes in group 3 can communicate with both groups 1 and 2.

Alfatafta et. al [9] conducted a study of the impact of partial network partitions on modern systems. They found that the majority of the studied failures are silent but cause catastrophic impact, such as data loss and corruption. The failures were also easy to manifest once a partition occurs, were deterministic and were caused by isolating a single node. Finally, they report that the majority of failures caused are due to design flaws, indicating that such network failures were unexpected by system designers.

Alfatafta et. al [9] report that modern approaches toward designing schedulers and resource managers are inherently vulnerable to partial partitions. Partial partitions can lead to resource unavailability, double execution of tasks that may violate semantic guarantees,

and complete system pause until the network recovers.

Consider a scenario where a partial partition that causes the scheduler to be isolated from one of the nodes. While the affected node can still reach the rest of the cluster, the scheduler will think the node has failed and will reschedule the tasks running on the affected node on other cluster nodes. This leads to double, potentially concurrent, execution of those tasks. Double execution can corrupt shared state (e.g., data on HDFS) or confuse clients. MapReduce [5], Mesos [6] and Elasticsearch [2] are examples of systems which have experienced this before.

2.2 Scheduling in Data Analytics Systems

We focus our discussion on resource management for data analytics platforms such as Apache Spark [69]. We choose Spark because it built the fundamental ideas for distributed data analytics and is widely adopted in the industry. We note that other data processing systems such as Apache Flink [3], Apache Storm [1], and Apache Hadoop[4] adopt an architecture that is similar to Spark’s. Consequently, our insights about partial network partitioning and our solutions are applicable beyond Spark.

Apache Spark is a data analytics system that can process large datasets on large clusters. Spark performs this efficiently by processing tasks in parallel across multiple computers. Spark does this by using Resilient Distributed Datasets (RDDs), a data structure that allows performing in-memory computations.

Figure 2.2 shows the Spark system architecture consisting of three components [59]: driver, cluster manager, and worker nodes. A worker node typically runs multiple executors, each of which can run one or more application-level tasks. A typical application life cycle is as follows. First, a client submits a job to Spark which creates a driver program. The driver is responsible for launching the application’s tasks, monitoring their progress, collecting the final results, and returning these results to the client. Driver works with the cluster manager (1) to allocate nodes for the application. Once resources are allocated, Spark schedules application tasks on allocated nodes (2). Workers nodes host the executor process which runs the application program on that node. During an execution, worker nodes may exchange data (3). The cluster manager continuously monitors the nodes through heartbeats (4). When execution is complete, the driver program collects the results and returns them to the client.

While Spark comes with a basic resource manager, called Spark Standalone manager, Spark is compatible and readily supports integration with other resource managers including

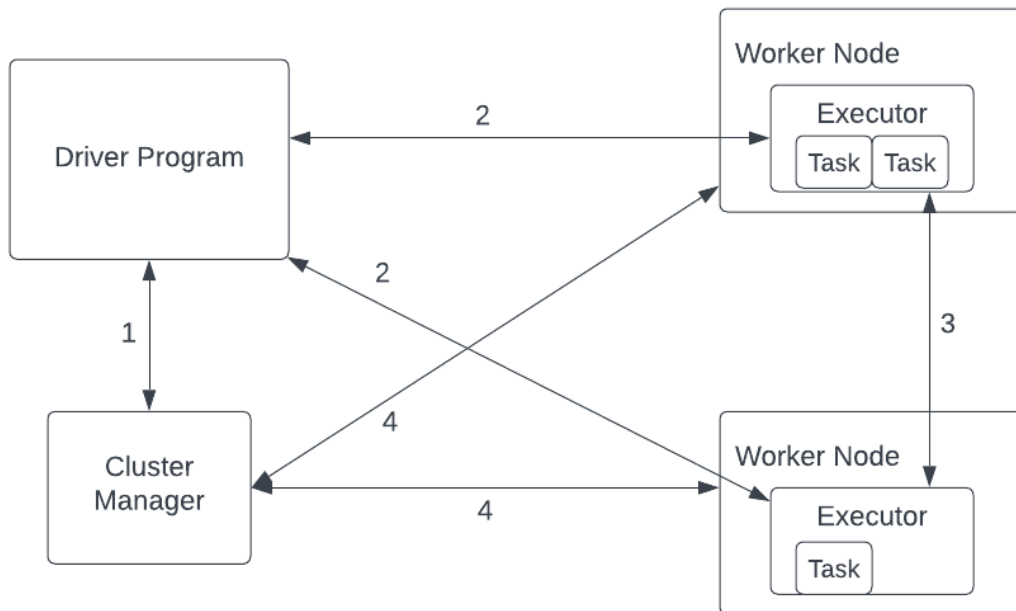


Figure 2.2: Apache Spark

Mesos [37] and Kubernetes [47]. Resource managers are also known as cluster manager in the literature, we use these two terms interchangeably.

Spark Standalone is a basic cluster manager and runs the basic functionalities of providing the driver with just enough resources to run an application. The following section discuss Mesos and Kubernetes resource managers.

2.2.1 Mesos

Apache Mesos [37] is a cluster manager that handles resource management in large distributed environments. In Spark, Mesos has daemons (a.k.a. Mesos agents) running on cluster nodes. The agents monitor a node’s CPU and memory resources and inform the cluster manager of available resources at that node. The cluster manager aggregates information about available resources in a cluster and offers those resources to the application driver. The driver can decline or accept all or some of the offered resources. The driver then starts executors on allocated nodes, then assigns tasks to executors.

2.2.2 Kubernetes

Kubernetes [47] is a container management platform that can manage large clusters. Kubernetes uses containers to run its application. Containerization is a deployment process in which an application is packaged with all of its dependencies and isolated environment and it is able to run on any infrastructure. In Kubernetes, containers run inside pods, which are the smallest and simplest unit of a Kubernetes system. A pod is a group of one or more containers sharing resources.

Figure 2.3 shows the architecture of Kubernetes. Kubernetes components are split into two planes: control and data. The control plane consists of the components that are running on the master node, which are controller manager, api server, etcd, and scheduler. Components running on the data plane run on all the worker nodes in the cluster and run the kubelet

etcd is a highly-available key value store that is used to store all data in the cluster. The API Server validates and configures data for the api objects.

Figure 2.3 presents an overview of how these components interact with one another when a general application runs. The client contacts the API server and requests to deploy an application. The API server notifies the controller manager of this new deployment. The manager creates the new pod for the new application and notifies the API server. The API Server notifies the scheduler of the unassigned pod. The scheduler selects a node to host the pod, updates the pod assignments, and informs the API server of the assignment. The API Server notifies the Kubernetes daemon (kubelet) that is running on the target node of the pod that has been assigned to it. The kubelet uses the container runtime to start the pod and launch the application.

When running Spark with Kubernetes, Spark creates a driver program and asks Kubernetes to deploy it. Kubernetes follow the aforementioned steps to deploy a pod and the run the driver program. Once the driver pod starts, the driver submits a request to the API server to create pods for the executors. The scheduler schedules the executor pods and the application starts. When the application completes, the executor pods terminate and the driver pod collects the logs. The driver pod is eventually terminated.

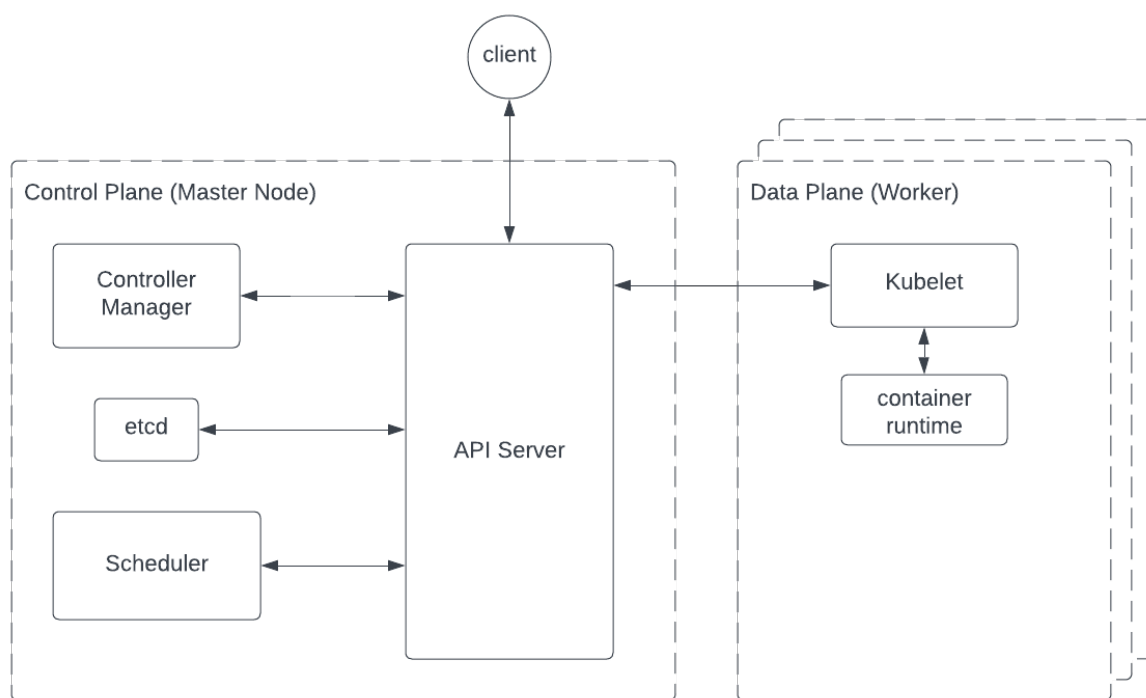


Figure 2.3: K8s Deployment

Chapter 3

The Impact of Partial Network Partitions on Scheduling

In this chapter, we look into the impact of partial network partition on scheduling in modern cluster managers. We empirically study three cluster managers and analyze their resiliency to partial network partitions.

3.1 Methodology

We follow an empirical approach to study the impact partial partitions have on cluster managers. We deploy the target system of our study and run a word counting application. We then inject a partial network partition between the cluster nodes. We examine the outcome of the execution and measure the application execution time. Finally, we study the code to understand why systems fail under certain partitions and why their performance is effected.

Workload. Analyzing the impact of partial network partitions on cluster managers requires an application which submits tasks to them. For our analysis, we use Spark as it is compatible with all our chosen cluster managers. The workload we chose to run on Spark is an instance of the simple WordCount application, which comes bundled with it [69]. For this analysis, we ran the WordCount application with an input file of size of 5GB and 10GB.

WordCount uses the MapReduce [21] paradigm to count of the number of occurrences of each word in a large corpus. MapReduce processes data in parallel by splitting it into chunks

across nodes and grouping the results for each word. This results in shuffling intermediate results amongst nodes. We selected this workload because it is simple, data intensive, and includes the typical stages of any data analytics application (i.e., map, shuffle and reduce).

The general workflow of the WordCount application is as follows: a job submission first results in the creation of a driver program by the cluster manager. The driver works with the cluster manager to allocate executors on worker nodes. After this, the input data is divided between executors who run mapping tasks [69]. The executors exchange intermediate results among themselves after the map stage, known as a "shuffle". Once this data transfer is complete, the executors perform reduce tasks and send their results to the driver. To simplify system behavior, we run a single executor on each worker node.

Approach. We follow an empirical approach to study the impact partial partitions have on cluster managers. We start by deploying the target cluster manager on our nodes and run the WordCount application, recording its execution time and outcome. We then inject a partial network partition between the chosen cluster nodes, before running WordCount again to record the execution time and outcome. We compare these results to examine the differences in application behavior and performance when the cluster manager experiences a partial network partition. Finally, we study the code to understand why cluster managers fail in certain scenarios and outline why their performance is affected.

Considering the system architecture of data analytics systems (Figure 2.2), we identify four categories of partial partitions which can impact them. For each of the scenarios below, communication is only broken between the specified nodes i.e. they can still communicate with the remaining nodes in the cluster. The categories are:

1. Scenario W-W: a partial partition between worker nodes that disrupts the communication between two or more worker nodes.
2. Scenario D-W: a partial partition between the application driver and one or more worker nodes.
3. Scenario M-W: a partial partition between the cluster manager and one or more worker nodes.
4. Scenario D-M: a partial partition between the application driver and the cluster manager.

Within each category, we consider two scenarios related to when the partial partitioning fault occurs: before the application starts (pre-existing) and while the application is running (mid-application). We experimented with these scenarios for each of the four categories, leading to eight test cases for each cluster manager.

3.2 Spark Standalone Cluster Manager

When using the Spark Standalone cluster manager, the cluster manager and the application driver can only be run on the same node. As we examine partial partitions at node granularity, D-W and M-W partitions have similar effects and are considered as one. This also excludes D-M partitions, as they are now on the same node, leaving us with two unique kinds of partitions: W-W and D-W. Figure 3.1 shows a diagram of a Spark cluster operating in standalone mode with three worker nodes.

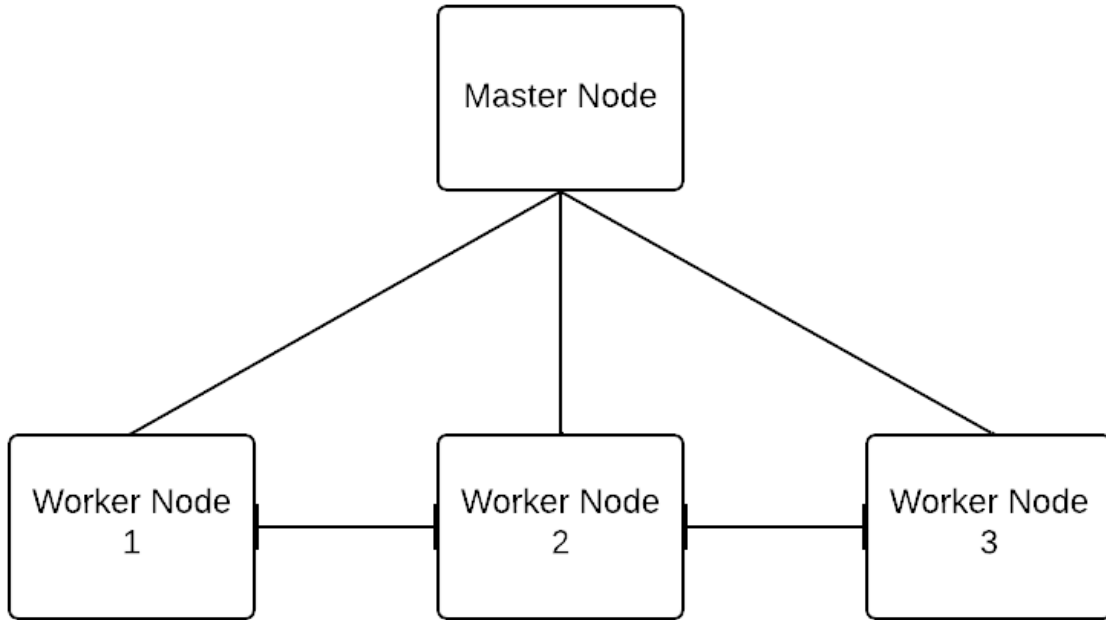


Figure 3.1: Spark Standalone

W-W Partition. In this scenario we inject a single partial partition between two worker nodes. Our experiments show that the impact of the partition is the same regardless if the partition was injected before the application starts or during its execution. Workers typically communicate during the shuffle step of the application to exchange intermediate results. During this step, executors running map tasks send out their intermediate results to the executor running the relevant reduce task. If a partial partition breaks the communication between two workers, this data transfer fails and the destination worker waits indefinitely for

the intermediate results. Workers do not report this communication problem to the driver, instead choosing to retry the data transfer at regular intervals. Thus, neither the driver nor the spark standalone cluster manager are able to detect this problem and the application freezes until the partial partition is healed. Our experiments show that the impact of the partition is the same regardless if the partition was injected before the application starts or during its execution.

D-W Partition. In this scenario, we inject a single partial partition between the node hosting the driver program and a worker node. The impact of this partition changes depending on when it is injected.

Pre-existing Partition. After worker nodes have been allocated to run executors, the driver tries to establish a connection with the nodes to launch them. If a driver and a worker node are partitioned, the driver cannot communicate with the worker node and assumes that it has crashed. Consequently, the driver does not assign any tasks to the partitioned worker, distributing all its executors among the remaining worker nodes. This reduces the number of nodes available to run the application, degrading application performance.

Mid-application partition. If a partition is injected after the driver assigns tasks to a worker, the driver will not be able to communicate with the partitioned worker. The driver declares this worker to have failed and reassigns its tasks to a different, potentially busy, worker. Thus, despite the partitioned worker being able to finish executing its assigned tasks and exchanging data with other worker nodes, its results are unused. In addition, task reassignment and re-execution on the new worker adds delays to the application execution. Thus, this partition results in performance degradation as well as wasted computation.

3.3 Mesos

When running Spark with Mesos, The driver runs on a separate node from the Mesos cluster manager (a.k.a. Mesos Master). The worker nodes continue to run on 3 separate nodes. Figure 3.2 is a representation of what this cluster would look like.

The Mesos master continuously offers all available resources to the driver. When a client submits an application to the driver, the driver selects the nodes for the application from the list of available resources it receives from Mesos. The driver runs the executors on the allocated worker nodes, then the application is executed.

W-W Partition. The timing of the partitions, whether before or during an application run results in similar outcomes. Similar to the Spark Standalone W-W scenario, cutting

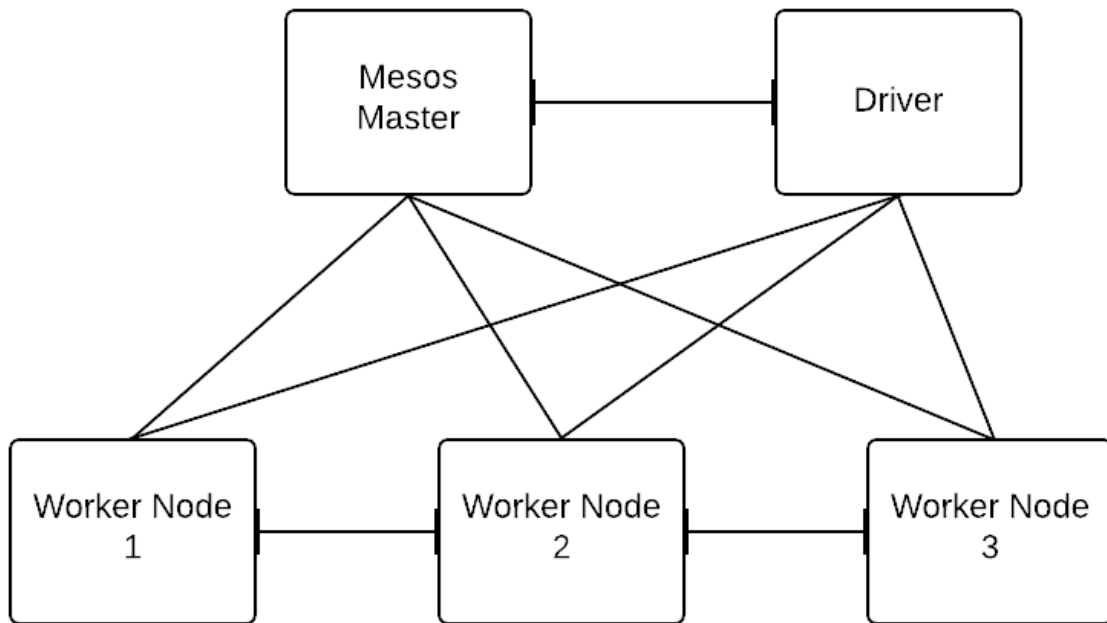


Figure 3.2: Mesos as a CM.

the communication between the worker nodes disrupts the shuffle stage. This causes the application to stop until the partition is fixed.

D-W Partition. In this scenario, we inject a single partial partition between the node hosting the driver program and a worker node. While the driver can not communicate with the partitioned worker, the cluster manager is not aware of this connectivity issue.

Pre-existing Partition. The cluster manager offers all available nodes to the driver program, including the node partitioned from the driver. The driver driver program rejects any node that it cannot reach. This results in the application running on fewer nodes, which slows down the application.

Mid-application partition. When a partial partition happens between the driver and a worker, the driver suspects that the partitioned worker crashed and reassigns it's workload to other workers. This fault leads to using fewer nodes and causes re-execution of tasks that are running on the partitioned node.

M-W Partition. In this scenario, we inject a single partial partition between the

Mesos master node and a worker node. The impact of this partition varies depending on its time of occurrence.

Pre-existing Partition. If the cluster manager can not reach a node, it assumes that the node crashed and will not offer it to future allocation requests. The partitioned node sits idle until the partition is fixed, which reduces the available cluster resources.

Mid-application partition. If the partition occurs after the manager allocates a node to a driver, the application program continues operating normally. This is because the driver program can communicate with the partitioned node. An issue arises if the driver program needs more resources, in this case the cluster manager will not offer the partitioned node to the driver.

M-D Partition. In this scenario, we inject a single partial partition between the Mesos master node and the node hosting the driver program. The impact of this partition varies depending on its time of occurrence.

Pre-existing Partition. If a partition happens before an application starts, the driver will not receive resources from the cluster manager and the program will not run.

Mid-application partition. If the partition occurs during an application execution, the driver program continues running uninterrupted. This is because once the resources had been offered by the cluster manager, the driver program does contact the cluster manager again. However, if a failure manifests where the driver will need additional resources, it will not be able to contact the cluster manager.

3.4 Kubernetes

As with Mesos, The Kubernetes Scheduler and the driver run on separate nodes. Figure 3.3 is a representation of this cluster.

Kubernetes is different compared to the other previous cluster managers as task scheduling is performed by the Kubernetes scheduler and not left up to the discretion of the Spark application driver. When an application is created, Spark notifies Kubernetes that it has a driver program that requires scheduling. The Kubernetes scheduler then schedules the driver. Once the driver starts, it request the Kubernetes scheduler to the schedule the executor pods to nodes. Once they've been scheduled, the application is able to start and complete execution. When the tasks completes, the executor pods terminate and are cleaned up.

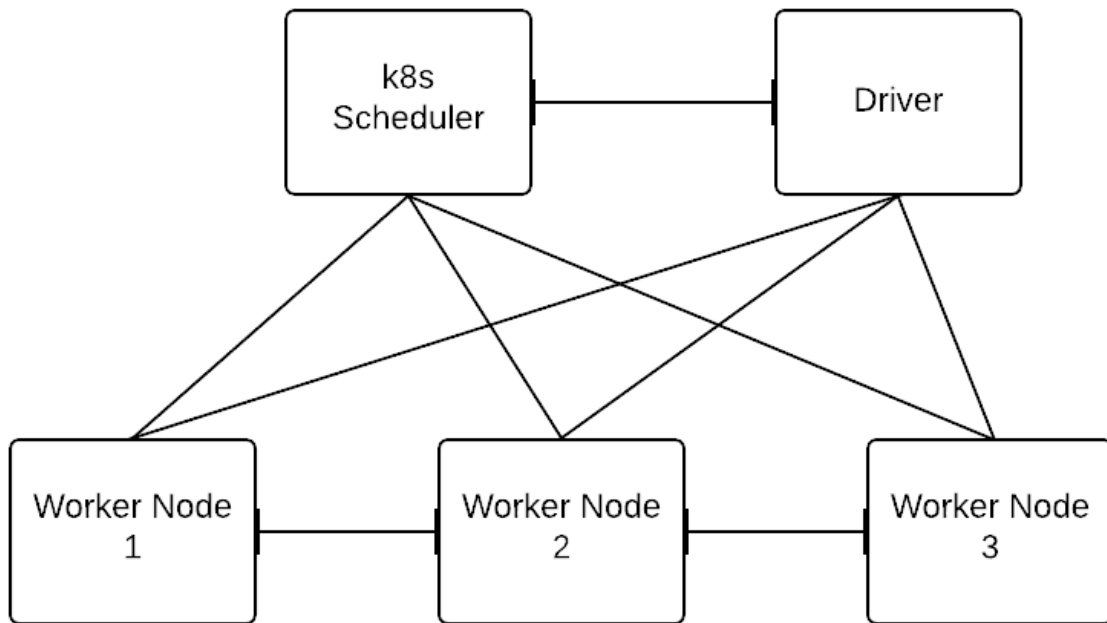


Figure 3.3: K8s as a CM

W-W Partition. A partial partition between two workers in Kubernetes has a different consequence than Mesos and Spark Standalone. The impact is the same regardless if the partition happens before or during an application run. During the shuffle stage the connection between the impacted workers times out. The Scheduler recognizes that there is an issue among the worker nodes and the workload is then reassigned to a new executor pod. Unfortunately, the scheduler is not aware of the network partition consequently, it is possible that the scheduler reassigns the task on a pod on a partitioned nodes, sometimes on the original node. This will cause the data transfer to fail and for the scheduler to reassign the tasks again. This adds a significant delay to the execution time. The same impact happens if a partition occurs after the application program starts.

D-W Partition. In this scenario, we inject a single partial partition between the node hosting the driver pod and a worker node hosting its executor pods. The impact of this partition varies depending on when it occurs.

Pre-existing partition. If a driver can not reach some of its assigned executor pods, the driver will assume that they have crashed and will not assign tasks to them. This leads to

an increase in execution time due to the application running on fewer resources.

Mid-application partition. If the partition occurs after the application starts, the driver requests Kubernetes to kill the executor. Kubernetes creates a new executor and assigns it to a new node, then the driver restarts the task on the new node. Kubernetes can potentially restart the task on a node that is partitioned from the driver and this would repeat the whole scenario. This increases the application execution time.

M-W Partition. In this scenario, we inject a single partial partition between the Kubernetes master node and a worker node. The impact of this partition varies depending on when it occurs.

Pre-existing partition. If the scheduler can not reach a node, it assumes that it crashed and does not use it in future executions. This leaves the partitioned node idle until the partition heals.

Mid-application partition. If a network partition occurs between the scheduler and a worker node after the program has begun execution, kubernetes will not recognize that the node is unreachable and the program will continue to finish executing uninterrupted. If the driver program were to need new resources, kubernetes will then recognize that the node is partitioned and then assume that the node has crashed and no pods are assigned to the partitioned node. Consequently, this partition scenario does not effect the current running tasks but reduces the resources available for future tasks.

M-D Partition. In this scenario, we inject a single partial partition between the Kubernetes master node and the node hosting the driver pod. The impact of this partition varies depending on when it occurs.

Pre-existing partition. A network partition between the Kubernetes scheduler and a driver before an application starts prohibits an application from starting at all, this is because Kubernetes can assign the driver pod to a node that is partitioned, the pod will then sit idle waiting for the assigned node to contact the scheduler and run the driver. The assigned node will never begin executing the driver program because it cannot contact the scheduler. The application does not start until the partition is fixed.

Mid-application partition. A network partition between the Kubernetes Scheduler and the driver after the application starts does not influence the execution time. The Kubernetes scheduler will not recognize that it has lost a node until after the driver program has finished executing. The application is unaffected because the driver has already acquired the resources it needs. The only difference is that the users will be unable to access the logs of the driver program pod.

Table 3.1: Summary of the impact of partial partitions. H indicates that the application stops making progress until the partial partition is fixed. P indicates that partial partition causes performance degradation. - indicates that there is no impact on the running application. R indicates one repeat of task execution. We discuss TAS in Section 6.

System	Partial Partition	Impact	
		Pre-App	Mid-App
Spark Standalone	W-W	H	H
	D-W	P	P
Mesos	W-W	H	H
	D-W	P	P
	M-W	P	-
	M-D	H	-
Kubernetes	W-W	P	P
	D-W	P	P
	M-W	P	-
	M-D	H	-
TAS	W-W	-	R
	D-W	-	R
	M-W	-	-
	M-D	-	-

3.5 Summary

Table 3.1 shows a summary of the impact of partial partitions on different systems under different scenarios. We note that partial partitions often lead to negative impact including pausing an application or reducing its performance. The performance degradation can be significant, our results in Section 6 show that partial partitions can increase the applications execution time by eleven times.

It is evident that a partial network partition between system components can have a significant impact on scheduling systems, regardless of the cluster manager you may use. In some instances, we observe that the application fails to complete execution, while in other cases, a partial partition causes significant delays. We notice that a Worker to

Worker partition leads to a significant impact in all system we study. Additionally, a partial partition between driver and worker always leads to performance degradation.

We note that during W-W partitions, Spark Standalone and Mesos get stuck while Kubernetes does not. This is due to Kubernetes scheduler working differently than other cluster managers. In Kubernetes, when two nodes can not complete the shuffle stage the scheduler restarts one of the tasks on another pod. The application does not pause and eventually completes execution. However it is important to note that in the case of M-D partition, the application program will not start with in Mesos and Kubernetes and is not reported as a problem by any system.

Our analysis shows that the main reason that causes these negative impacts, is that the component that schedules tasks is not aware of the of the new topology in the cluster, i.e., it does not know which nodes can communicate. Consequently, in pre-app cases, the scheduler allocates nodes that are not all-to-all connected, leading to many of the failure scenarios in Table 3.1. In scenarios in which a partial partition occurs during an application run, the scheduler either does not react, leaving the application stuck until the partition heals, or reacts and allocates new nodes for the application. In the latter case, it still allocates nodes without considering the connectivity in a cluster, and may allocate nodes impacted by the partition to the application. This causes the application to fail again and request different nodes. This iteration continues until the schedule, by mere luck, allocates nodes that are all-to-all connected to the application. In the following chapter we explore building a topology aware scheduler that can avoid the shortcomings of current scheduling techniques.

Chapter 4

Topology Aware Scheduling

We propose the Topology-Aware Scheduler (TAS) approach to address the shortcomings of the current resource management techniques. TAS mitigates the impact of partial partitions by incorporating the information about cluster connectivity into scheduling.

TAS extends Kubernetes scheduling approach. We choose Kubernetes because it is widely adopted in the industry and is the system that best handles partial partition faults among the state-of-the-art solutions (Chapter 3). To encourage adoption, a primary goal of TAS is to require minimal changes to Kubernetes. Furthermore, in Kubernetes, unlike the other scheduler that task the application driver to schedule tasks, the Kubernetes scheduler is the only entity making scheduling decisions for the driver and tasks. This makes it easier to implement TAS.

The main insight behind TAS is that when allocating multiple nodes to an application, the scheduler should allocate nodes that are fully connected. In order to achieve this, TAS uses a connectivity graph to identify nodes that are fully connected.

Figure 4.1 shows the design of TAS. TAS has three main components: the connectivity monitoring overlay, the scheduler, and the scheduler Metadata Service. The *connectivity monitoring overlay* monitors the connectivity in a cluster and provides a *connectivity matrix* representing the connectivity between nodes. The *scheduler* uses this connectivity matrix to allocate nodes for an application. The *scheduler metadata service* keeps track of the locations of previously allocated pods, per application.

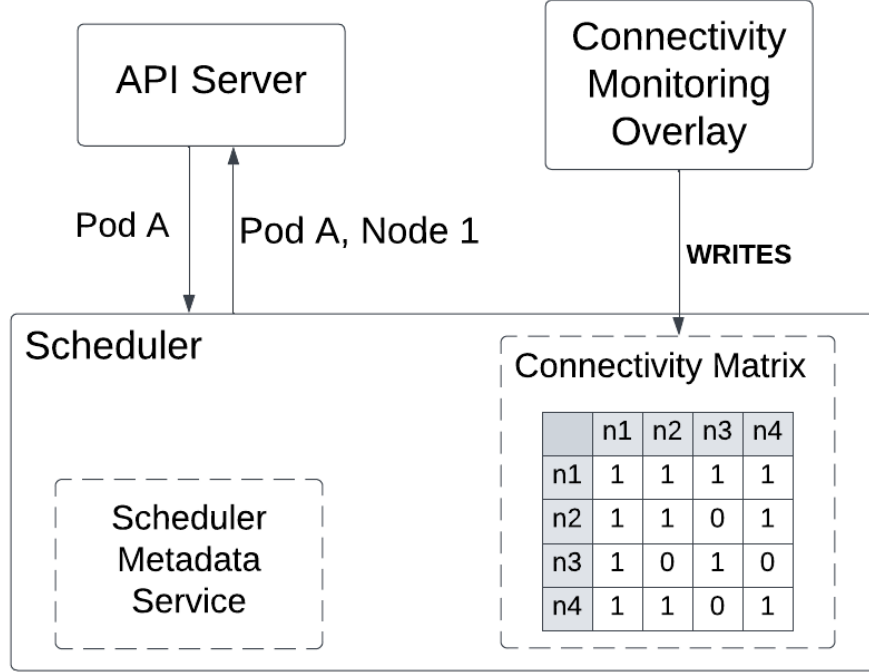


Figure 4.1: TAS Design.

4.1 Connectivity Monitoring Overlay

To monitor cluster connectivity we run a *connectivity daemon* on each node in the cluster. Each daemon stores a *connectivity matrix* to representing the connectivity across the cluster. This connectivity matrix is mapped to a file local to the node, enabling it to be accessed by other processes running within the node. For instance, Figure 4.2 presents what a connectivity matrix of a 4 node cluster with a partial network partition would look like. Each row and column corresponds to a node's connectivity. Row/Column n1 correspond to Node 1's connectivity; seeing as node 1 is fully connected, they are all filled with 1s. Node 3 is disconnected from nodes 2 and 4 so cells (n3,n2), (n2,n3), (n4,n3) and (n3,n4) are all set to 0 indicating the lack of connection between these nodes. The TAS *scheduler* accesses the file stored by its local daemon and uses it when making scheduling decisions. The connectivity matrix in Figure 4.1 is an example of a matrix of a cluster of four nodes when experiencing a partial network partition.

	n1	n2	n3	n4
n1	1	1	1	1
n2	1	1	0	1
n3	1	0	1	0
n4	1	1	0	1

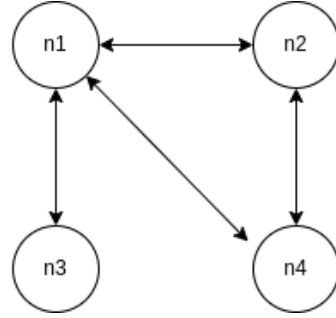


Figure 4.2: Connectivity Matrix of a 4 node cluster.

Each daemon sends a periodic heartbeat to all daemons in the cluster. The default heartbeat period is 200 ms. If a daemon misses three heartbeats from another node, it assumes that the node is unreachable and updates its connectivity matrix. Nodes continue to send heartbeats to disconnected nodes in case the connection is restored.

When a node detects a change in the connectivity matrix it sends its connectivity matrix to all other nodes in the cluster. When a daemon receives an connectivity matrix from another daemon, it updates its matrix and forward the updated matrix to its connections.

Daemons store the connectivity matrix to local file that can be mapped. The TAS schedule accesses the connectivity matrix of its local daemon and uses it to inform its scheduling decisions.

4.2 Scheduler Metadata Service

Each application submitted to Kubernetes is assigned a unique identifier (Application ID). Subsequent task requests include this identifier. Whenever the scheduler receives a task request, it queries the *scheduler metadata service* to inquire if any pods have previously been allocated to the requesting application. If no pods have previously been allocated, the scheduler creates a new entry mapping to the application ID within the scheduler metadata service. Anytime a pod is allocated, the scheduler metadata service for the application is updated with the pod name and node address.

As more pods are scheduled, the scheduler can query the Scheduler Metadata service to determine where previous pods were allocated. This, along with the connectivity matrix, gives the scheduler the information it needs to make scheduling decisions. When an

application terminates, the scheduler notifies the Scheduler Metadata service to delete the contents of that application using its ID.

4.3 Scheduler Design

The native Kubernetes scheduler (kube-scheduler) selects a node for a pod in two steps, filtering and scoring. During the filtering step, kube-scheduler finds the set of feasible nodes i.e. nodes with enough resources to run the pod. The scoring step assigns a score to each of the nodes. The scheduler assigns a score to each node based on the scoring rules set by the application. For instance, an application can prefer to schedule its pods on fewer nodes to reduce network overhead, or to disperse the pods on more nodes to reduce contention. The kube-scheduler then selects a feasible node with the highest score to run the pod.

TAS extends the current scheduling steps with two additional step. After filtering and scoring, TAS first queries the *scheduler metadata service* to identify the locations of previously allocated pods. TAS then uses the *connectivity matrix* to select a node that is fully connected to the nodes hosting previously allocated application pods.

After filtering and scoring, TAS first queries the *scheduler metadata service* to identify the locations of previously allocated pods. TAS then uses the *connectivity matrix* to select a node that is fully connected to the nodes hosting previously allocated application pods.

For subsequent allocations, TAS queries the metadata service to locate all the nodes that are hosting active pods for this application. It then uses the connectivity matrix to find new nodes which are fully-connected to *all* these previous nodes. Finally, TAS allocates the node which is most connected (i.e., has highest number of connections) and can be reached by all the nodes that are hosting active pods for the application and updates the metadata service. Whenever a pod is terminated, either by the application or Kubernetes, TAS removes the relevant entry from the metadata service.

Algorithm 1 shows the pseudocode of our scheduling algorithm and the following is the step by step explanation of the algorithm:

1. Read connectivity file and store contents in the *connectivityMatrix*. (line 2)
2. Get the *nodeList* that contains all the nodes and their score. (line 3)
3. If scheduling the first pod (i.e., the driver pod) (line 4-5):

Algorithm 1 Scheduling algorithm

```
1: procedure SELECTCONNECTEDNODE(pod, appID)
2:   connectivityMatrix  $\leftarrow$  Read from file
3:   nodeList  $\leftarrow$  GetFeasibleNodes()
4:   if pod is first pod then
5:     selectedNode  $\leftarrow$  MostConnected(nodeList, connectivityMatrix)
6:   else
7:     prevPods  $\leftarrow$  SchedulerMetadataService.Get(appID)
8:     selectedNode  $\leftarrow$  MostConnectedTo(prevPods, nodeList, connectivityMatrix)
9:   podAlloc  $\leftarrow$  pod, selectedNode
10:  SchedulerMetadataService.Set(appID, podAlloc)
11:  return podAlloc
```

- (a) Compare the nodes in *connectivityMatrix* and select the node with the highest connectivity.
 - (b) If all connectivity is the same, select the node with the highest score. Break ties in score by random selection.
4. If scheduling the worker pod (line 6-8):
- (a) Query Scheduler Metadata Service to get list of application pods and the nodes they are running at. Store the list in *prevPods* (line 7).
 - (b) Filter *nodeList* to identify nodes connected to all the nodes in *prevPods*.
 - (c) Among these filtered nodes, select the node with the highest connectivity. If two nodes have the same connectivity, select the node with the highest score. If connectivity and score are the same, select a node randomly.
5. Store the name of the selected node for worker pod. (line 9)
6. Update the Scheduler Metadata Service with the new pod allocation metadata. (line 10)
7. Return the pod information (line 11)

Every allocation has a complexity of $O(N^2)$ because we compare a new node with previously connected nodes to ensure connectivity.

When there are no partial partitions in the cluster, TAS allocations are identical to allocations by kube-scheduler. If there is no single node in the cluster with full connectivity, TAS chooses the node with the highest connectivity, regardless of its Kubernetes score. The reasoning behind this is that nodes have already gone past the filtering stage, and any node deemed feasible by the scheduler is a node with enough resources to run the program.

When a spark application runs, the driver is the always the first pod to get assignment. If we take our cluster in figure 4.2, our most fully connected node would node 1, and hence the driver program would be allocated to node 1.

In figure 4.2 we see that among the nodes that are connected to node 1, nodes 2 and 4 are the ones with the highest number of neighbors, consequently, a subsequent pod will be assigned to either 2 or 4. The application program will continue to run on the subset nodes of 1, 2, and 4. Which are all fully connected nodes and leads to least disruption in the application program life cycle.

Chapter 5

Implementation

We implement TAS in 180 line of code of go lang and the network monitor daemon in 441 lines of C++ code. We use Kubernetes v1.20.7. We base our implementation on the open source implementation of the Kubernetes default scheduler, kube-scheduler. We extended the kube-scheduler to integrate topology information when making scheduling decisions.

Connectivity daemons. We implemented the daemon using C++. We run a connectivity daemon on all the nodes in a cluster. A config file lists all the ip addresses of a cluster nodes. The daemon heartbeats all the nodes that are in the config file. This heartbeating is done over UDP packets with a default heartbeat period of 200ms.

Each daemon build an adjacency matrix to capture the connectivity in a cluster. A daemon stored the adjacency matrix into a file. The file is accessed by the TAS scheduler to get the connectivity information.

Scheduler. We implemented the TAS scheduler by doing small changes to the current Kubernetes schedule. In particular we re-implemented the `SelectHost()` function. The `SelectHost()` function is the core function of the Kubrnatess scheduler.

In our implementation of `SelectHosts`, called `SelectConnectedHosts`, we open and read the connectivity file that was created by gathering information from all the daemons running on the nodes in the cluster. We map the indices of the array to the node names that Kubernetes already has stored and are able to create a 2-dimensional adjacency matrix.

Chapter 6

Evaluation

In this chapter we empirically evaluate TAS performance by comparing it to the original Kubernetes scheduler. We evaluate its fault tolerance capability, and throughput with and without a partial network partition. We ran each experiment 10 times and report average and standard deviation.

Testbed. We conduct our experiments on a 5 node cluster hosted within the Cloudlab [25] Utah data center. We used c6525-25g nodes [65] each of which had 16 AMD EPYC cores, 125GB RAM, and 10 Gbps network connection. One of the nodes is always used to host the scheduler while the other 4 are used as worker nodes.

Workloads. We use the WordCount workload outlined in Section 3 for our evaluation. We also include TeraSort, a workload which comes bundled with Spark [69]. TeraSort sorts a large amount of data via the MapReduce paradigm and involves large network transfers during its shuffle phase.

6.1 Fault Tolerance

We evaluate TAS' fault tolerance under the four partial partition scenarios presented in Section 3. Table 3.1 shows the result of our evaluation and compares TAS to the other schedulers. Our evaluation show that TAS avoids the negative impacts of the previous schedulers and application running using TAS never gets stuck or experience significant performance degradation.

In all pre-existing partition scenarios, TAS completely eliminates the negative impacts imposed by them. This is because TAS scheduling ensures that for a given application, all

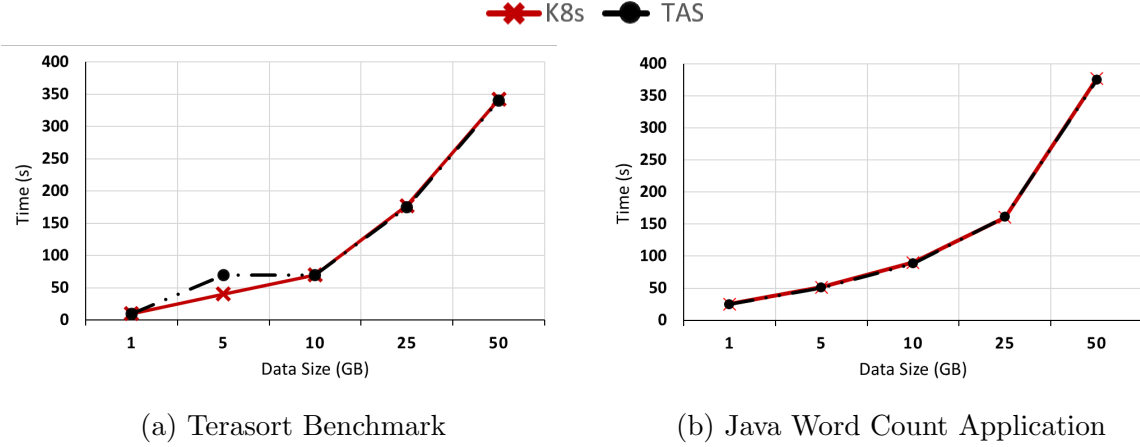


Figure 6.1: Benchmarking TAS and Kubernetes baseline scheduler against an increasing workload. The standard deviation for these experiments is less than 6%.

application pods are allocated on fully-connected nodes. During mid-application partitions, under M-W and M-D partition scenarios, TAS behaves similar to the kube-scheduler, ensuring that the application is not impacted by the partition.

An application running with TAS experiences a small performance degradation when a partial partition occurs during an application run between two workers (W-W) or a worker and the driver (D-W). Under these two scenarios, a partition disrupt data transfer between workers or prohibits the driver to access the worker results. In both these scenarios, TAS selects a new node to run the pod. The new selected node is fully connected to the other nodes running the application. The performance degradation is due to the re-execution of a task on a new node. Our evaluation later in this section show that this degradation is minor compared to that of the base Kubernetes scheduler under the same partition circumstances.

6.2 Overhead Evaluation

To measure the overhead imposed by TAS we measure application execution time under a partition-free scenario, when it uses TAS and the kube-scheduler. We ran both WordCount and TeraSort for this experiment, varying input data sizes from 1 GB to 50 GB.

Figures 6.1a and 6.1b plot the application execution time against input data size, when running Terasort and WordCount respectively. Our results show that under partition-free

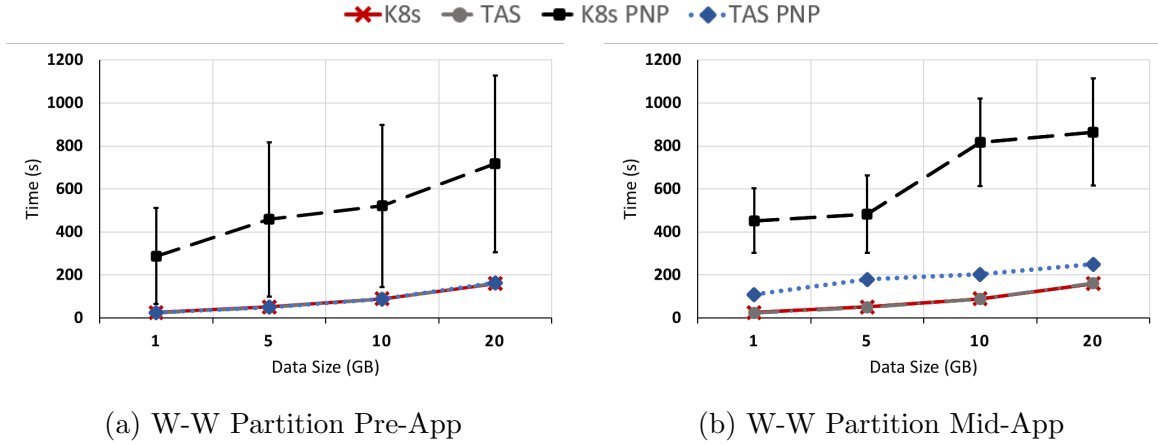


Figure 6.2: Average execution time of word count application under W-W partition. K8s, TAS, and TAS PNP have a standard deviation less than 3%. The figure shows the standard deviation for K8s PNP.

scenarios, TAS' performance is comparable to the kube-scheduler operating within 2% of execution time on average.

6.3 Performance Under Partial Partitions

We compare the performance of TAS against the kube-scheduler under the four partial partitions categories outlined in Section 3: W-W, D-W, M-W, and M-D. Additionally we vary the time of partition injection, evaluating both pre-existing and mid-application partitioning scenarios. We use the WordCount workload for this analysis. The results for TeraSort are similar and have been omitted from the paper for clarity. Each of the figures compares the following:

- *K8s* - kube-scheduler under a partition-free scenario.
- *TAS* - TAS under a partition-free scenario.
- *K8s PNP* - kube-scheduler experiencing the specified partial network partition.
- *TAS PNP* - TAS experiencing the specified partial network partition.

W-W Partition. Figures 6.2a and 6.2b show the performance under the W-W partition scenario. *K8s PNP* suffers a significant performance degradation in both pre-existing and mid-application partitioning scenarios. Application execution time with *TAS PNP* is faster than *K8s PNP* by $4 - 11\times$ on average. *K8s PNP* also causes large variance in application execution time, as shown by the error bars.

A W-W partition disrupts the communication between two executor pods during the shuffle stage. Once Kubernetes detects this problem, it tries to reallocate one of the pods. Unfortunately, it is possible that the new worker node chosen for the pod is also impacted by the partial partition, causing the communication to fail again and Kubernetes to allocate a new pod on to a node. This process repeats until Kubernetes, by luck, allocates a node that is fully connected to all application nodes.

Pre-existing. On the other hand, WordCount with TAS experiences no performance degradation during pre-existing W-W partitions (Figure 6.2a). *TAS PNP* has execution times similar to those of *TAS* and *K8s* under partition-free scenarios. This is because TAS uses the information about cluster connectivity to schedule the application on a fully connected subset of nodes.

Mid-application. During mid-application partitions (Figure 6.2b), application execution time with *TAS PNP* is $3.5\times$ faster than *K8s PNP* on average and only $1.75\times$ slower than that of *K8s*. When TAS detects a problem between two workers, it allocates a new worker. It selects a worker that is fully connected to the application’s active pods and the application resumes execution. Unlike Kubernetes, TAS restarts tasks only once.

D-W Partition. Figures 6.3a and 6.3b show the performance of TAS and Kubernetes under D-W partitions. *K8s PNP* suffers a significant performance degradation during both pre-existing (Figure 6.3a) and mid-application partitions (Figure 6.3b), with TAS being $4 - 7\times$ faster on average.

Pre-existing. With pre-existing partitions, K8s may still allocate the partitioned worker to the application to host executor pods. However, the driver will not assign any tasks to these pods and the program will execute with less resources, leading to a performance degradation. However, *TAS PNP* in this case performs identical to partition-free scenarios. This is because TAS uses the information about cluster connectivity to schedule the application on a fully connected subset of nodes.

Mid-application. When the partition occurs mid-application, the driver pod will request for a new executor pod. Kubernetes can potentially allocate the new pod onto the same or a different partitioned node, causing the application to fail again as outlined above. With *TAS PNP*, occurs with a minor delay due to the time taken before the driver program requests a new worker node from the scheduler. When the driver detects the problem and

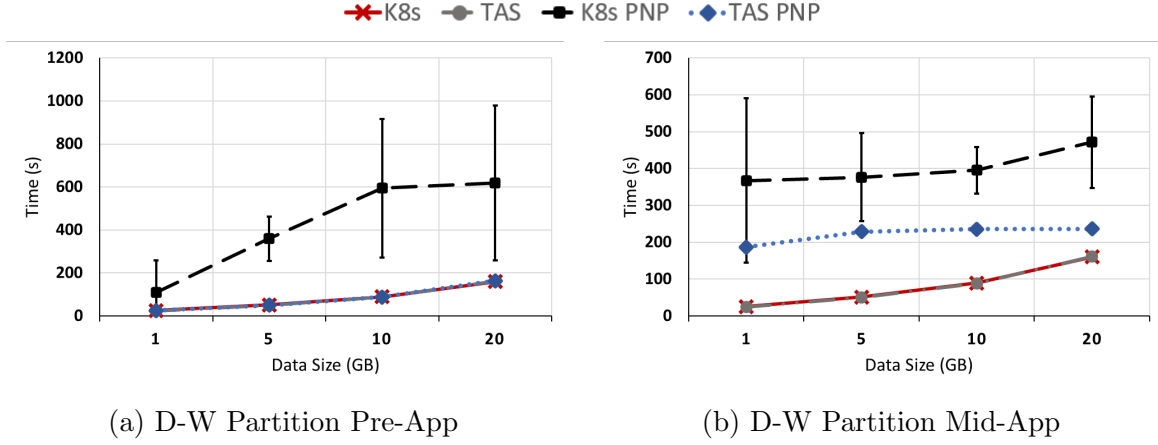


Figure 6.3: Average execution time of word count application under D-W partition. K8s, TAS, and TAS PNP have a standard deviation less than 3%. The figure shows the standard deviation for K8s PNP.

requests a new pod, TAS will create a pod and allocate it to a node that is fully connected to the previous nodes. Thus, TAS will only repeat the failed task once. In this scenario TAS is $2\times$ faster than *K8s PNP*.

It is important to note the high variance in *K8s PNP* results. Under W-W and D-W partitions (Figure 6.2), WordCount’s execution time with *K8s PNP* varies between 1 min 17 min with 5GB of data. This is due to the unpredictable number of trials Kubernetes makes until it allocates the impacted pod on a node that is fully connected with the rest of the application pods. TAS avoids this problem by using connectivity information to select a fully connected node.

M-W Partition. Figures 6.4a and 6.4b show the performance under M-W partial partitions.

Pre-existing. Figure 6.4 shows the performance under the M-W partial partition scenario. In the pre-app scenario (Figure 6.4a), Kubernetes suffers some performance degradation when a partition happens due to the fact that it will assign a pod to worker nodes that are unable to contact the scheduler and the worker nodes will fail to start. There is a delay until Kubernetes realizes this and restarts the task onto another worker node. TAS on the other hand will not allocate a pod onto a worker node that is partitioned from it and hence achieve a performance that is very similar to Kubernetes without a network partition.

Mid-application. In the M-W mid-app scenario (Figure 6.4b) all systems achieve similar performance. After an application starts the driver program does not contact the Kubernetes

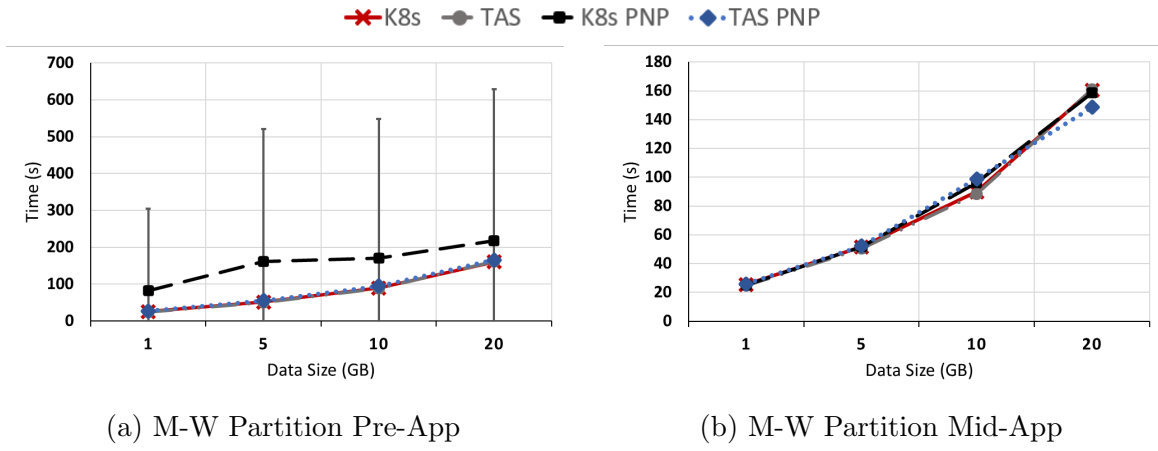


Figure 6.4: Average execution time of word count application under M-W partition. K8s, TAS, and TAS PNP have a standard deviation less than 5%. The figure shows the standard deviation for K8s PNP.

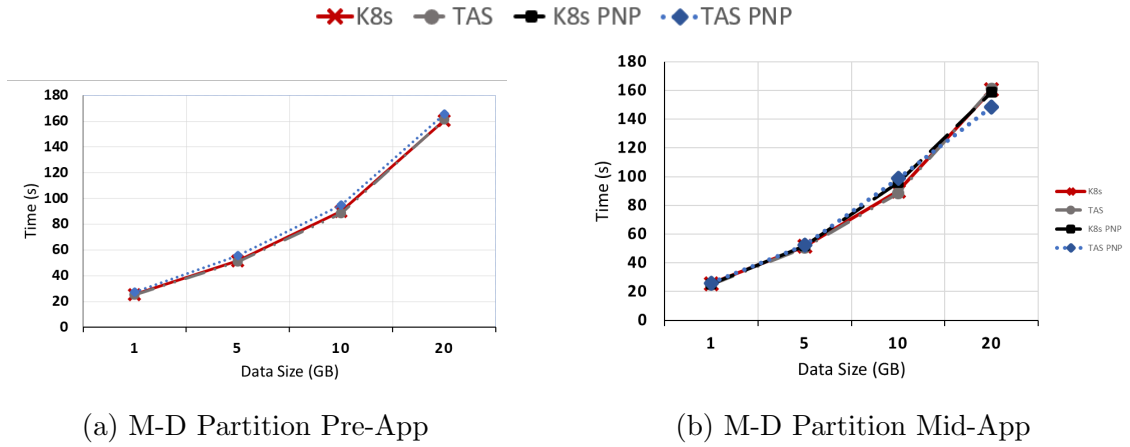


Figure 6.5: Average execution time of word count application under M-D partition.

scheduler (unless there is an error) and thus it is not impacted by the partial partition.

M-D Partition. Figure 6.5 shows the performance under the M-D partial partition scenario.

Pre-existing. In the pre-app scenario (Figure 6.5a) the application does not run at all because the driver pod is allocated to a node that is unable to communicate with the scheduler and the driver pod fails to start. TAS is able to run and execute successfully with execution times comparable to that of a run with no network partition because TAS allocates the driver pod onto a node that is fully connected to the TAS and the program is able to start executing.

Mid-application. In the M-D mid-app scenario (Figure 6.5b) all systems achieve similar performance. After an application starts the driver program does not contact the Kubernetes scheduler and it is not impacted by the partial partition.

Chapter 7

Related Work

Two research directions are related to our work: studies of faults and fault tolerant scheduling.

Study of Failures. NEAT [12] and NIFTY [10, 11] present a study of the impact of network partitions, including partial partitions, on distributed systems in general. NIFTY reports that 76.4% of the studied failures caused by partial network partitioning have catastrophic effects. The failures studied were found to be deterministic or have known time constraints. A majority of these failures (66.6%) require three or fewer events (other than the partial partition) to occur. Additionally, most failures (59.3%) do not require client access or only require clients to access one side of the partition. All of the studied failures can be triggered by a single-node partial partition, and design flaws account for most of the fixed bugs (59.3%). These studies motivated us to look into the problem that partial network partitions and its effects on scheduling in large scale data analytics systems.

A lot of work has been done to analyze failures in distributed systems. Some of these studies characterize a broader set of failures, but only for specific domain of systems and services, such as HPC [26, 50, 63], IaaS clouds [16], data-mining services [70], hosting services [57, 36], and data-intensive systems [48, 35, 61, 68]. While the rest of these efforts focused on specific component failures such as physical [67] and virtual machines [17], network devices [33, 66], storage systems [29, 42], software bugs [35], and job failures [48, 19, 31].

Bailis et al.[13] discuss publicly disclosed failures in deployed distributed systems and highlight the significance of network partitions as a possible cause of such failures. The authors emphasize the importance of taking network partitions into account during design

before they occur, as it is much easier to plan for partition failures ahead of time than it is to make changes to a complex system in a production environment.

Our work complements these efforts by studying the impact of partial partitions on scheduling for data analytic frameworks. Unlike previous efforts that relied on failure reports to study failures, we followed an empirical approach. We empirically studied the impact on the state-of-the-art schedulers and designed a new scheduling approach that use the insights from our study.

Fault Tolerant Scheduling

While there are many scheduling systems for data analytics workloads [46, 58, 69], they do not tolerate network partitions, partial or complete. Sparrow [58] and Falcon [46] expose task failures to the application and allow them to be handled in application specific format. Falcon uses a programmable switch to accelerate scheduling decisions, which further complicates their fault tolerance model.

Elzeki et al. [27] review the many different scheduling algorithms that can be applied in cloud computing and analyzes the methods often associated with task scheduling and how different characteristics such as the arriving rate of the tasks, cost of the task execution on each of the resource, cost of communication come into play when deciding which algorithm is best for your task.

Keivani et al. [45] study task scheduling in cloud computing frameworks and report that reliability, availability, and error handling require further improvements. They conclude that cloud services would improve if they deployed enhanced algorithms that take these parameters into account.

Fault tolerance is a major concern when it comes to the reliability and availability of a system. Bala and Chana [14] discuss the existing fault tolerance techniques in cloud computing based on their policies, tools used and research challenges but none of the techniques discussed refer to network partitions.

k8s-spark-scheduler-extender [56] claim that running Spark on Kubernetes can lead to resource issues and oversubscription, so they’ve created a Kubernetes Scheduler Extender that is designed to that a driver pod is only scheduled if there is enough space for all of its executors. Their scheduler extender is design specifically for spark applications and does not take into account network partitions.

Previous works [41, 49] explored using network characteristics and conditions to inform scheduling decisions with the focus on improving application overall performance. These efforts complement our efforts that focus on application fault tolerance.

TAS is the first scheduler that can tolerate partial network partitioning problems. TAS is the first scheduler to use cluster connectivity information and incorporate this information when making scheduling decisions. Our evaluation shows that this design approach is effective in tolerating partial network partitioning and compares to the state-of-the-art schedulers.

Chapter 8

Conclusion and Future Work

We conduct an in-depth and empirical study of the impact partial network partitioning has on modern schedulers for data analytics frameworks. Our study show that modern schedulers are vulnerable to partial network partitioning. Partial partitions often lead to severe negative impact such as application pause until the partition is fixed or a significant performance degradation.

Based on our insights from the aforementioned study, we proposed a new topology-aware scheduling technique (TAS) that can tolerate partial partitions. TAS incorporates cluster connectivity information in the process of scheduling. We implemented TAS by changing 180 lines of code in Kubernetes scheduler. Our evaluation shows that TAS can tolerate all scenarios of partial network partitioning and imposes negligible overhead.

The current design may not scale well with large cluster. The all-to-all heartbeating approach and exchanging an $O(N^2)$ connectivity matrix will not scale well for large clusters. We plan to investigate this and create a scalable design for TAS in our future work.

A second idea we plan to explore is build an overlay between the end nodes, and to forward traffic between end nodes around partial partitions. Our preliminary evaluation shows that this approach may increase network overhead on nodes along a path and may reduce overall performance for data intensive applications. We plan to study the application and network characteristics under which this approach is viable and can bring performance benefits.

Overall, our study highlights the importance of considering network failures and partial network partitions when designing and implementing scheduling techniques in modern distributed systems. By improving the fault tolerance of these systems, we can increase their reliability and performance, ultimately enabling more efficient data analysis.

References

- [1] Apache storm documentation. <https://storm.apache.org/releases/2.4.0/index.html/>. Accessed: April 2023.
- [2] Disconnect between coordinating node and shards can cause duplicate updates or wrong status code #9967. <https://github.com/elastic/elasticsearch/issues/9967>. Accessed: April 2023.
- [3] Flink architecture. <https://nightlies.apache.org/flink/flink-docs-release-1.17/docs/concepts/flink-architecture/>. Accessed: April 2023.
- [4] Hdfs architecture. <https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html>. Accessed: April 2023.
- [5] Mapreduce ticket 4832. <https://issues.apache.org/jira/browse/MAPREDUCE-4832>. Accessed: April 2023.
- [6] Mesos-1529: Handle a network partition between master and slave. <https://issues.apache.org/jira/browse/MESOS-1529>. Accessed: April 2023.
- [7] Summary of the amazon ec2 and amazon rds service disruption in the us east region, Apr 2011.
- [8] Kubernetes scheduler documentation, Dec 2022.
- [9] Mohammed Alfatafta, Basil Alkhatib, Ahmed Alquraan, and Samer Al-Kiswany. Toward a generic fault tolerance technique for partial network partitioning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 351–368. USENIX Association, November 2020.

- [10] Mohammed Alfatafta, Basil Alkhatib, Ahmed Alquraan, and Samer Al-Kiswany. Toward a generic fault tolerance technique for partial network partitioning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 351–368. USENIX Association, November 2020.
- [11] Basil Alkhatib, Sreeharsha Udayashankar, Sara Qunaibi, Ahmed Alquraan, Mohammed Alfatafta, Wael Al-Manasrah, Alex Depoutovitch, and Samer Al-Kiswany. Partial network partitioning. *ACM Trans. Comput. Syst.*, dec 2022. Just Accepted.
- [12] Ahmed Alquraan, Hatem Takruri, Mohammed Alfatafta, and Samer Al-Kiswany. An analysis of network-partitioning failures in cloud systems. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI’18, page 51–68, USA, 2018. USENIX Association.
- [13] Peter Bailis and Kyle Kingsbury. The network is reliable: An informal survey of real-world communications failures. *Queue*, 12:20–32, 07 2014.
- [14] Anju Bala and Inderveer Chana. Fault tolerance-challenges, techniques and implementation in cloud computing. *International Journal of Computer Science Issues*, 9, 01 2012.
- [15] Md. Faizul Bari, Raouf Boutaba, Rafael Esteves, Lisandro Zambenedetti Granville, Maxim Podlesny, Md Golam Rabbani, Qi Zhang, and Mohamed Faten Zhani. Data center network virtualization: A survey. *IEEE Communications Surveys & Tutorials*, 15(2):909–928, 2013.
- [16] Theophilus Benson, Sambit Sahu, Aditya Akella, and Anees Shaikh. A first look at problems in the cloud. HotCloud’10, page 15, USA, 2010. USENIX Association.
- [17] Robert Birke, Ioana Giurgiu, Lydia Y. Chen, Dorothea Wiesmann, and Ton Engbersen. Failure analysis of virtual and physical machines: Patterns, causes and characteristics. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 1–12, 2014.
- [18] Eric Brewer. Lessons from giant-scale services. *Internet Computing, IEEE*, 5:46 – 55, 08 2001.
- [19] Xin Chen, Charng-Da Lu, and Karthik Pattabiraman. Failure analysis of jobs in compute clouds: A google cluster case study. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 167–177, 2014.

- [20] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56:74–80, 2013.
- [21] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI’04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, CA, 2004.
- [22] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Voshall, and Werner Vogels. Dynamo: Amazon’s highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, oct 2007.
- [23] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Voshall, and Werner Vogels. Dynamo: Amazon’s highly available key-value store. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles, SOSP ’07*, page 205–220, New York, NY, USA, 2007. Association for Computing Machinery.
- [24] Pamela Delgado, Florin Dinu, Anne-Marie Kermarrec, and Willy Zwaenepoel. Hawk: Hybrid datacenter scheduling. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference, USENIX ATC ’15*, page 499–510, USA, 2015. USENIX Association.
- [25] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1–14, Renton, WA, July 2019. USENIX Association.
- [26] Nosayba El-Sayed and Bianca Schroeder. Reading between the lines of failure logs: Understanding how hpc systems fail. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12, 2013.
- [27] Omar Elzeki, Magdi Rashad, and Mohamed Abu Elsoud. Overview of scheduling tasks in distributed computing systems. *International Journal of Soft Computing and Engineering*, 2:470–475, 01 2012.
- [28] Avriella Floratou, Ashvin Agrawal, Bill Graham, Sriram Rao, and Karthik Ramasamy. Dhalion: Self-regulating stream processing in heron. *Proc. VLDB Endow.*, 10(12):1825–1836, aug 2017.

- [29] Daniel Ford, François Labelle, Florentina I. Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. Availability in globally distributed storage systems. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, page 61–74, USA, 2010. USENIX Association.
- [30] Daniel Ford, François Labelle, Florentina I. Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. Availability in globally distributed storage systems. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, page 61–74, USA, 2010. USENIX Association.
- [31] Peter Garraghan, Paul Townend, and Jie Xu. An empirical failure-analysis of a large-scale cloud computing environment. In *2014 IEEE 15th International Symposium on High-Assurance Systems Engineering*, pages 113–120, 2014.
- [32] Seth Gilbert and Nancy Lynch. Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, jun 2002.
- [33] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: Measurement, analysis, and implications. *SIGCOMM Comput. Commun. Rev.*, 41(4):350–361, aug 2011.
- [34] Ramesh Govindan, Ina Minei, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. Evolve or die: High-availability design principles drawn from google’s network infrastructure. 2016.
- [35] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patanapanake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. What bugs live in the cloud? a study of 3000+ issues in cloud systems. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, page 1–14, New York, NY, USA, 2014. Association for Computing Machinery.
- [36] Haryadi S. Gunawi, Mingzhe Hao, Riza O. Suminto, Agung Laksono, Anang D. Satria, Jeffry Adityatama, and Kurnia J. Eliazar. Why does the cloud stop computing? lessons from hundreds of service outages. SoCC '16, page 1–16, New York, NY, USA, 2016. Association for Computing Machinery.
- [37] Roger Ignazio. *Mesos fundamentals*, page 58–62. Manning, 2018.
- [38] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle,

- Stephen Stuart, and Amin Vahdat. B4: Experience with a globally-deployed software defined wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 3–14, New York, NY, USA, 2013. Association for Computing Machinery.
- [39] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a globally-deployed software defined wan. *SIGCOMM Comput. Commun. Rev.*, 43(4):3–14, aug 2013.
 - [40] jepsen-io / jepsen Jepsen-Io. Jepsen-io/jepsen: A framework for distributed systems verification, with fault injection, 2015.
 - [41] Jingjie Jiang, Shiyao Ma, Bo Li, and Baochun Li. Symbiosis: Network-aware task scheduling in data-parallel frameworks. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9, 2016.
 - [42] Weihang Jiang, Chongfeng Hu, Yuanyuan Zhou, and Arkady Kanevsky. Are disks the dominant contributor for storage failures? a comprehensive study of storage subsystem failure characteristics. *ACM Trans. Storage*, 4(3), nov 2008.
 - [43] Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, Desislava Dimitrova, Matthew Forshaw, and Timothy Roscoe. Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 783–798, Carlsbad, CA, October 2018. USENIX Association.
 - [44] Konstantinos Karanasos, Sriram Rao, Carlo Curino, Chris Douglas, Kishore Chali-
parambil, Giovanni Matteo Fumarola, Solom Heddaya, Raghu Ramakrishnan, and
Sarvesh Sakalanaga. Mercury: Hybrid centralized and distributed scheduling in large
shared clusters. In *Proceedings of the 2015 USENIX Conference on Usenix Annual
Technical Conference*, USENIX ATC '15, page 485–497, USA, 2015. USENIX Associa-
tion.
 - [45] Arghavan Keivani and Jules-Raymond Tapamo. Task scheduling in cloud computing:
A review. In *2019 International Conference on Advances in Big Data, Computing and
Data Communication Systems (icABCD)*, pages 1–6, 2019.
 - [46] Ibrahim Kettaneh, Sreeharsha Udayashankar, Ashraf Abdel-hadi, Robin Grosman,
and Samer Al-Kiswany. Falcon: Low latency, network-accelerated scheduling. In

Proceedings of the 3rd P4 Workshop in Europe, EuroP4'20, page 7–12, New York, NY, USA, 2020. Association for Computing Machinery.

- [47] Kubernetes Documentation. <https://kubernetes.io/docs/home/>, 2023. [Online; accessed 24-Mar-2023].
- [48] Sihan Li, Hucheng Zhou, Haoxiang Lin, Tian Xiao, Haibo Lin, Wei Lin, and Tao Xie. A characteristic study on failures of production distributed data-parallel programs. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 963–972, 2013.
- [49] Zhuozhao Li, Haiying Shen, and Ankur Sarker. A network-aware scheduler in data-parallel clusters for high performance. *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2018.
- [50] Yinglung Liang, Yanyong Zhang, A. Sivasubramaniam, M. Jette, and R. Sahoo. Bluegene/l failure analysis and prediction models. In *International Conference on Dependable Systems and Networks (DSN'06)*, pages 425–434, 2006.
- [51] Rupak Majumdar and Filip Niksic. Why is random testing effective for partition tolerance bugs? *Proc. ACM Program. Lang.*, 2(POPL), dec 2017.
- [52] A. Markopoulou, G. Iannaccone, S. Bhattacharyya, Chen-Nee Chuah, Y. Ganjali, and C. Diot. Characterization of failures in an operational ip backbone network. *IEEE/ACM Transactions on Networking*, 16(4):749–762, 2008.
- [53] Justin Meza, Tianyin Xu, Kaushik Veeraraghavan, and Onur Mutlu. A large scale study of data center network reliability. In *Proceedings of the Internet Measurement Conference 2018*, IMC '18, page 393–407, New York, NY, USA, 2018. Association for Computing Machinery.
- [54] Tony Mills. Bnx2 cards intermittantly going offline.
- [55] Andrew Montalenti. Kafkapocalypse: A postmortem on our service outage, Mar 2015.
- [56] Chrisbattarbee onursatici, rkaram. k8s-spark-scheduler. <https://github.com/palantir/k8s-spark-scheduler>, 2023.
- [57] David Oppenheimer, Archana Ganapathi, and David A. Patterson. Why do internet services fail, and what can be done about it? In *4th USENIX Symposium on Internet Technologies and Systems (USITS 03)*, Seattle, WA, March 2003. USENIX Association.

- [58] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 69–84, New York, NY, USA, 2013. Association for Computing Machinery.
- [59] Jean-Georges Perrin and Rob Thomas. *Spark in action*. Manning Publications Co., 2020.
- [60] Rahul Potharaju and Navendu Jain. When the network crumbles: An empirical study of cloud network failures and their impact on services. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, New York, NY, USA, 2013. Association for Computing Machinery.
- [61] Ariel Rabkin and Randy Howard Katz. How hadoop clusters break. *IEEE Software*, 30(4):88–94, 2013.
- [62] Andrew Schrank and Josh Whitford. The anatomy of network failure. *Sociological Theory*, 29(3):151–177, 2011.
- [63] Bianca Schroeder and Garth A. Gibson. A large-scale study of failures in high-performance computing systems. *IEEE Transactions on Dependable and Secure Computing*, 7(4):337–350, 2010.
- [64] Rachee Singh, Muqet Mukhtar, Ashay Krishna, Aniruddha Parkhi, Jitendra Padhye, and David Maltz. Surviving switch failures in cloud datacenters. *SIGCOMM Comput. Commun. Rev.*, 51(2):2–9, may 2021.
- [65] The CloudLab Team. Cloudblab documentation. <https://docs.cloudlab.us/hardware.html>. Accessed: April 2023.
- [66] Daniel Turner, Kirill Levchenko, Alex C. Snoeren, and Stefan Savage. California fault lines. *Proceedings of the ACM SIGCOMM 2010 conference*, 2010.
- [67] Kashi Venkatesh Vishwanath and Nachiappan Nagappan. Characterizing cloud computing hardware reliability. SoCC '10, page 193–204, New York, NY, USA, 2010. Association for Computing Machinery.
- [68] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed Data-Intensive systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 249–265, Broomfield, CO, October 2014. USENIX Association.

- [69] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*, pages 15–28, 2012.
- [70] Hucheng Zhou, Jian-Guang Lou, Hongyu Zhang, Haibo Lin, Haoxiang Lin, and Tingting Qin. An empirical study on quality issues of production big data platform. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 17–26, 2015.