CASPR: Connectivity-Aware Scheduling for Partition Resilience

Sara Qunaibi*, Sreeharsha Udayashankar*, Samer Al-Kiswany*#
*University of Waterloo, Canada
#Acronis Research, Canada
{squnaibi, s2udayas, alkiswany}@uwaterloo.ca

***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 shortcomings of the state-of-the-art schedulers, we design CASPR, a connectivity-aware scheduler. CASPR incorporates the current network connectivity information when making scheduling decisions to allocate fully connected nodes for a given application. CASPR effectively hides partial partitions from applications. Our evaluation of a CASPR prototype shows that it can tolerate partial network partitions, as well as eliminate application halting or significant loss of performance.

***Index Terms*—cloud computing, computer networks, network partitions, fault tolerance**

I. Introduction

Modern large-scale applications in domains such as data analysis [1, 2], stream processing [3, 4], and online services [5], 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 [6, 7, 8] have shown that network partitions, particularly partial network partitions, pose special challenges for system design. Partial partitions are network partitions that divide the network into three groups (say G1, G2, and G3) such that two groups (say G1 and G2) are disconnected, whereas G3 is able to communicate with all nodes in the cluster. Studies of failure reports [7] in production systems show that this network fault leads to catastrophic failures, often due to system design flaws.

In this work, 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 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 [9], Mesos [10], and the Spark native scheduler [1]. We identify the stages in an application's execution that can be impacted by a network partition, and then use fault injection to empirically evaluate the impact of a partition occurring 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× longer execution time.

By studying the designs of these state-of-the-art schedulers, we identify a key shortcoming. Modern schedulers assume that network connectivity is transitive that is 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 Connectivity-Aware Scheduler for Partition Resilience (CASPR). CASPR uses cluster connectivity information to augment its scheduling decisions. When a partial network partition occurs, CASPR identifies the nodes that are still connected, and allocates fully connected nodes to new applications. If a partial partition happens during application execution, CASPR adjusts an application's allocation to ensure that it runs on a subset of nodes that are fully connected.

We have implemented CASPR by extending the Kubernetes scheduler. Our evaluation with standard data analytics workloads shows that CASPR eliminates all the negative impacts of partial partitioning. With CASPR, applications never get stuck, and their performance is not degraded significantly. Compared to state-of-the-art schedulers CASPR improves the application runtime by up to 7× under partial network partitions.

The rest of this paper is organized as follows. We present an overview of the design of modern schedulers in Section II. Section III presents the methodology and results of our study. We present CASPR's design in Section IV, offer our evaluation in Section VI, and conclude in Section VIII. The source code is publicly available on GitHub [11].

II. Background

A. Network Partitioning

A network partition is a fault that causes nodes to be divided into groups that cannot communicate with each another. These network failures can have a significant impact on service availability and performance [6, 12]. Network partitioning faults are common: Google reports experiencing 40 network

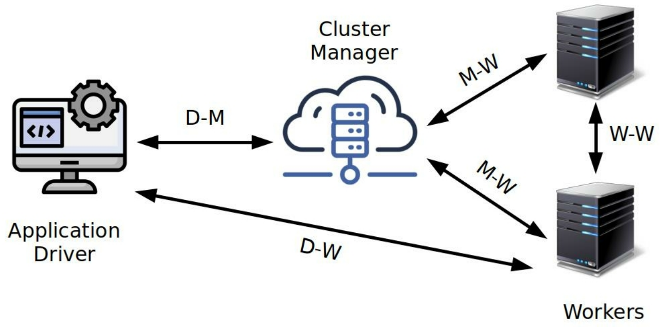


Fig. 1: Data analytics system architecture.

partitions in two years [13], Microsoft reports that 70% of the downtime is caused by network partitions [14], and Turner et al. found that network partitions occur almost once every 4 days in the California-wide CENIC network [15].

Network partitions have a wide array of causes. They can manifest in geo-replicated systems because of a loss of connectivity between data centers [15], while switch failures can cause network partitions within a data center [14]. Network partitions caused by correlated failures are common [15, 16], often caused by system-wide maintenance tasks [13, 14]. On a single node, NIC or software failures can partition a node that may host multiple Virtual Machines [17].

Network partitions are among the most complex failures as they cannot be masked by the transport layer and impact system semantics. Taking network partitions into account complicates system design, fault analysis, and debugging. The CAP [18] theorem sheds light on the theoretical limits of availability and consistency in the presence of network partitions.

Partial Network Partitioning. The most complex network partitions are partial network partitions. Alfatafta et al. [7] report that modern approaches toward designing schedulers and resource managers are inherently vulnerable to partial partitions. Partial partitions can lead to resource unavailability, double the execution of tasks that may violate semantic guarantees, and complete system pause until the network recovers.

Consider a scenario where a partial partition isolates the scheduler from one of the nodes. Although the affected node can still reach the rest of the cluster, the scheduler 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 the shared state (e.g., data on HDFS) or confuse clients [19]. MapReduce [20], Mesos [21] and ElasticSearch [22] are examples of systems that have experienced such failures.

B. Scheduling in Data Analytics Systems

We focus our discussion on resource management for data analytics platforms such as Apache Spark [1]. We choose Spark because it built the fundamental ideas for distributed data analytics and is widely adopted. As other data processing systems such as Apache Flink [23], Apache Storm [24], and Apache Hadoop [25] adopt an architecture similar to Spark's, our insights and solutions are applicable beyond Spark.

The Spark architecture (Figure 1) consists of three components [26]: 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 Spark application life-cycle is as follows. First, a client submits a job to Spark that creates the driver program. The driver is responsible for launching executors to run application tasks, monitoring their progress, collecting the final results, and returning these results to the client. The driver works with the cluster manager to acquire worker nodes to run executors.

During execution, executor processes may exchange data. The cluster manager continuously monitors the worker nodes through heartbeats whereas the driver heartbeats its executors to monitor their progress. When execution is complete, the driver program collects the results and returns them to the client.

Although Spark comes with a basic resource manager, called the Spark Standalone manager, Spark supports other resource managers including Mesos [10] and Kubernetes [9]. We use resource manager and cluster manager interchangeably.

1) Spark Standalone: Spark Standalone is a basic cluster manager and has just the functionalities required to provide drivers with enough resources to run their applications.

2) Mesos: Apache Mesos [10] is a cluster manager that handles resource management in large distributed environments. 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 the resources available within that node. The cluster manager aggregates information about available resources from all agents in a cluster and offers those resources to application drivers. The driver can decline or accept the offered resources. The driver then launches executors on allocated nodes.

3) Kubernetes: Kubernetes [9] is a container management platform for large clusters. Kubernetes uses containers to run applications. Containerization is a deployment process in which an application is packaged with all of its dependencies, enabling it to run on any Linux Kernel. 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.

Kubernetes components are split into two planes: control and data. The control plane consists of the components responsible for infrastructure management, such as the controller manager, API server, and scheduler. Control plane components run on the master node. Components running on the data plane run on all worker nodes, such as kubelets.

When an application runs, it begins with the client contacting the API server with an application deployment request. The API server notifies the controller manager of this new deployment. The manager creates a pod for the new application and sends an acknowledgement to the API server. Then, the API Server contacts the scheduler with the unassigned pod.

The scheduler selects a node to host the pod, updates the pod assignments, and informs the API server of the assignment. Finally, the API server contacts the Kubernetes daemon (kubelet) running on the node assigned by the scheduler. The kubelet uses the information within the container 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 follows the aforementioned steps to deploy a pod and 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 chooses nodes to host these 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.

III. The Impact of Partial Network Partitions on Scheduling

In this section we present the first in-depth empirical study of the impact of partial network partitions on scheduling systems. The goal of this study is to understand the impact partial partitions have on state-of-the-art schedulers and to identify the design flaws that cause these failures. We use the findings from our study to inform the design of CASPR (Section IV).

A. Methodology

Target Cluster Managers. We examine the impact of partial partitions on Kubernetes, Mesos, and the Spark Standalone cluster manager.

Workload. The workload we chose to run is the simple WordCount application, which comes bundled with Spark [1]. We run the WordCount application with input files of sizes 5GB and 10GB.

WordCount uses the MapReduce [2] paradigm to count the number of occurrences of each word in a large corpus. Spark processes data in parallel by splitting it into chunks across nodes and grouping the results for each word. This results in shuffling intermediate results among nodes. We selected this workload because it is simple, data intensive, includes the typical stages of any data analytics application (i.e., map, shuffle and reduce), and requires all Spark components in Figure 1 to communicate.

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. Then, the input data are divided between executors who run mapping tasks [1]. The executors exchange intermediate results among themselves, 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 that 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 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 architecture of data analytics systems (Figure 1), we identify four categories of partial partitions that can impact them. For each of the categories, communication is only broken between the specified nodes i.e. they can still communicate with the remaining nodes in the cluster. The categories are as follows:

1) **`W-W Partition:`** A partial partition between two or more worker nodes.
2) **`D-W Partition`**: A partial partition between the application driver and one or more worker nodes.
3) **`M-W Partition`**: A partial partition between the cluster manager and one or more worker nodes.
4) **`D-M Partition`**: 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.

B. Spark Standalone Cluster Manager

When using the Spark Standalone cluster manager [1], 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. 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.

`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 of whether 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, they retry the data transfer at regular intervals. As both workers appear to be working correctly to the driver and cluster manager, neither is able to detect this problem. Thus, the application halts until the partial partition is healed.

`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 of 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 launches executors on a worker, the driver will not be able to communicate with these executors. The driver declares these executors failed and relocates them to a different, potentially busy, worker. Thus, despite executors on the partitioned worker being able to finish their assigned tasks and exchange data with other executors, their results are unused. In addition, executor relocation and re-execution on the new worker node add delays to the application execution. Thus, this partition results in performance degradation and wasted computation.

C. Mesos

When running Spark with Mesos [10], the Mesos master continuously offers a list of all available resources to the application driver. The driver selects the nodes it needs for executors from this list. The driver launches executors on the chosen worker nodes, and then assigns tasks to them. Within this section, we discuss the impact of each category of partitions when using Mesos.

`W-W Partition.` In this scenario, we inject a single partial partition between two worker nodes. This fault impacts the shuffle stage. This causes the application to halt until the partition is fixed. The timing of the partition's occurrence does not affect its impact.

`D-W Partition.` In this scenario, we inject a single partial partition between the node hosting the driver program and a worker node. Although the driver cannot communicate with executors hosted on the partitioned worker, the cluster manager is not aware of any problems as it can communicate with both the worker node and the driver. The impact of this partition varies depending on when it occurs.

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

Mid-application partition. When a partial partition happens between the driver and a worker, the driver suspects that executors on the partitioned worker have crashed and relocates them to other worker nodes. This fault leads to using fewer nodes and causes the re-execution of tasks that are running on the partitioned node, leading to performance degradation.

`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 when it occurs.

Pre-existing partition. If the cluster manager cannot reach a worker node, it assumes that the node has crashed and will not offer it during future resource offers. The partitioned node is unused until the partition is healed, reducing the overall available cluster resources, causing performance degradation.

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 still communicate with executors on the partitioned node. In case the driver program needs more resources, the cluster manager's resource offers will exclude the partitioned node.

`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 any resource offers from the cluster manager and the program will not run.

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

D. Kubernetes

Kubernetes is different when compared to the previous cluster managers, as task scheduling is performed by the Kubernetes scheduler and not by the 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 pod. Once the driver starts, it requests the Kubernetes scheduler to schedule the executor pods to nodes. Following this, the application starts.

`W-W Partition.` In this scenario we inject a single partial partition between two worker nodes. A partial partition between two workers impacts Kubernetes differently when compared to Mesos and Spark Standalone. The impact is the same regardless of when the partition occurs.

During the shuffle stage, the communication between the impacted workers times out. The scheduler recognizes that there is an issue among the executor pods, and the workload is reassigned to a new executor pod. Unfortunately, as the scheduler is not aware of the network partition, it can inadvertently assign the new executor pod onto the same or another partitioned node. This will cause repeated data transfer failure and executor reassignment, adding significant delays and variance to the application execution time.

`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 cannot reach some of its assigned executor pods, the driver will assume that they have crashed and will not assign tasks to them. The application's tasks are redistributed among the remaining available executor pods. This leads to application performance degradation as the application runs on fewer resources.

Mid-application partition. If the partition occurs after the application starts, the driver requests Kubernetes to kill the executor pod. Kubernetes creates a new executor pod and assigns it to a new node, after which the driver restarts the task on the new pod. Kubernetes can potentially assign the new executor pod to a node that is partitioned from the driver. As a result, driver will be unable to assign tasks to the executor pod, leading to fewer resources and performance degradation.

`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. After a number of consecutive heartbeats are missed, the scheduler declares a worker node as dead. Once a node is declared dead, it is not used in future allocations. However, if a partition occurs before this period, fresh pods may be allocated onto this node. As the node is partitioned from the scheduler, these pods will be unable to start. Thus, in both cases, performance is degraded due to the reduction of available cluster resources.

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 execute uninterrupted. If the driver program were to need new resources, Kubernetes will then recognize that the node is partitioned, assume that the node has crashed and will not assign pods to the partitioned node. Consequently, this partition scenario does not affect currently 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 prevents an application from starting at all. This is because Kubernetes can assign the driver pod to a partitioned node, which will then wait 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 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. However, users will be unable to access the logs of the driver program pod.

TABLE I: Summary of the impact of partial partitions. Pre-App refers to partitions that occur before the application starts. Mid-App refers to partitions that occur while an application is running.

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	-
CASPR	W-W	-	R
	D-W	-	R
	M-W	-	-
	M-D	-	-

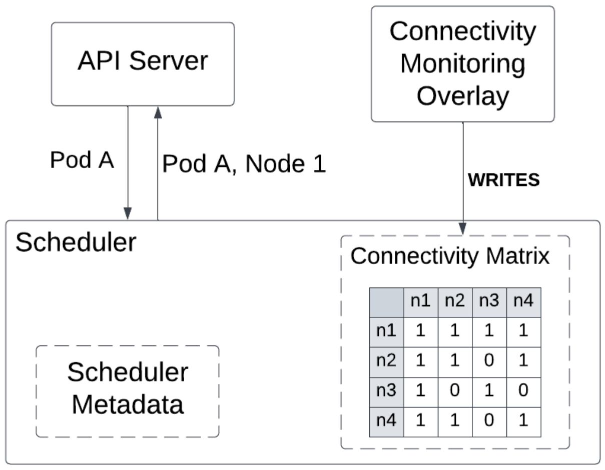
E. Summary

Table I shows a summary of the impact of partial partitions on different systems in different scenarios. `H` indicates application halting, `P` indicates performance degradation and `R` indicates a single re-execution or repetition. We note that partial partitions often lead to a severe negative impact, regardless of the cluster manager used. In some instances, we observe that the application is halted until the partition is resolved whereas in others, there is unbounded re-assignment and re-execution of tasks. This behavior causes significant performance degradation for running applications, up to a $11\times$ increase in application execution time (Section VI).

Cluster manager behavior during partial partitions differs as well. For instance, during W-W partitions, Spark Standalone and Mesos cause the application to halt whereas Kubernetes does not due to the Kubernetes scheduler functioning differently from the other cluster managers. In other cases, such as M-D partitions, the application program will not start in Mesos or Kubernetes, and no system can detect the problem.

In scenarios with pre-existing partitions, the scheduler may allocate a group of nodes that are not all-to-all connected to applications, leading to many of the failure scenarios in Table I. In scenarios with mid-app partitions, the scheduler either does not react, leaving the application hanging, or reacts by allocating new nodes for the application. In the latter case, these allocations do not consider cluster connectivity, causing the application to fail repeatedly. Because this process repeats until the scheduler allocates a fully connected group by pure chance, it can result in large performance degradation, as shown in Section VI.

Our analysis shows that these problems are a result of scheduling components not being connectivity-aware; that is,



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

Fig. 2: CASPR's Design.

they do not know which nodes are connected to each other. In the following chapter we discuss how CASPR overcomes the shortcomings of current scheduling techniques.

IV. Connectivity-Aware Scheduling for Partition Resilience

We propose the Connectivity-Aware Scheduling for Partition Resilience (CASPR) approach to address the shortcomings of current techniques. CASPR mitigates the impact of partial partitions by incorporating information about cluster connectivity into scheduling decisions. CASPR builds off Kubernetes' scheduling approach because it is the best among the state-of-the-art schedulers when handling partial network partitions. However, as one of the design goals of CASPR was to enable easy adoption, it is generic and easily extendable to other scheduling systems.

The main insight into CASPR is that when allocating multiple nodes to an application, the scheduler should allocate nodes that are fully connected. In order to achieve this, CASPR uses a connectivity graph to identify nodes that are fully connected at the time of allocation.

Figure 2 shows the design of CASPR, which has three main components: the connectivity monitoring overlay, the scheduler, and the scheduler metadata. 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* keeps track of the locations of previously allocated pods, per application.

A. Connectivity Monitoring Overlay

To monitor cluster connectivity we run a *connectivity daemon* on each node in the cluster. Each daemon stores a *connectivity matrix* representing 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. The CASPR *scheduler* accesses the file stored by its local daemon and uses it when making scheduling decisions. Figure 2 shows the connectivity matrix of a 4-node cluster experiencing a partial network partition. A value of `1` in the matrix indicates that the pair of nodes can communicate with each other, whereas a value of `0` indicates otherwise.

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, it assumes that the node is unreachable and updates its connectivity matrix. Daemons continue to attempt sending heartbeats to daemons on disconnected nodes, to detect when the connection is restored.

When a daemon detects a change in its connectivity matrix it sends its matrix to all other daemons in the cluster. When a daemon receives a matrix from another daemon, it updates its own matrix and forwards this updated matrix to its connections.

B. Scheduler Metadata

Each application submitted to Kubernetes is assigned a unique identifier (application ID). Subsequent pod allocation requests include this identifier. Whenever the scheduler receives a pod allocation request, it queries the *scheduler metadata* to inquire whether any pods have been allocated to the requesting application. If no pods have been allocated, the scheduler creates a new entry mapping to the application ID within the scheduler metadata.

Anytime a pod is allocated, the scheduler metadata for the application is updated with the pod's name and node address. When a pod is terminated, its entry is removed from the application metadata. When an application is terminated, the scheduler deletes the application mapping from within the scheduler metadata.

Before allocating fresh pods to an application, the scheduler queries the scheduler metadata to determine where previous pods were allocated. This, along with the connectivity matrix, gives the scheduler the connectivity information it needs to make scheduling decisions.

C. 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. Finally, the `kube-scheduler` selects the feasible node with the highest score to run the pod.

CASPR adds two additional steps to this. After filtering and scoring, CASPR queries the *scheduler metadata* to identify the locations of previously allocated pods. CASPR then uses the *connectivity matrix* to select a node that is fully connected to the nodes hosting previously allocated application pods.

Application drivers typically request that multiple pods run executors. The native Kubernetes client sends these requests one pod at a time, and each request includes the unique application ID. If CASPR is scheduling the first pod for an application, it consults the connectivity matrix to choose the

Algorithm 1 Scheduling Algorithm

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

most highly connected node among all filtered nodes. For subsequent allocations, CASPR 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 that are fully-connected to all previous nodes.

CASPR always allocates the node that is most connected (i.e., has the most connections) and can be reached by all the nodes that are hosting active pods for the application. Whenever a pod is terminated, either by the application or Kubernetes, CASPR removes the relevant entry from the metadata.

Algorithm 1 shows the pseudocode of our scheduling algorithm, whose steps are detailed below:

1) Read connectivity file the local *connectivity daemon* generates. (line 2)
2) Get the *nodeList* containing all feasible nodes and their score. (line 3)
3) If scheduling the first pod (lines 4-5),
   a) Compare the nodes in *connectivityMatrix* and select the node with the highest connectivity.
   b) If all connectivities are equal, select the node with the highest score. Break ties in score by random selection.
4) If scheduling a subsequent pod (lines 6-8),
   a) Query scheduler metadata to obtain *prevPods*, the list of active application pods and nodes hosting them (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. Break ties in connectivity by selecting the node with higher score. If connectivity and score are the same, select a node randomly.
5) Update scheduler metadata and return allocated pod information. (lines 9-11)

Every allocation has a complexity of $O(N^2)$, where N is the number of cluster nodes, because we compare a new node with all previously allocated nodes to ensure connectivity.

When there are no partial partitions in the cluster, CASPR allocations are identical to allocations by `kube-scheduler`. If no single node in the cluster has full connectivity, CASPR chooses the node with the highest connectivity regardless of its Kubernetes score because nodes have already gone past the filtering stage, and any node the scheduler deems feasible is a node with enough resources to run the program.

We illustrate the CASPR allocation protocol with an example. Consider the connectivity matrix shown in figure 2. When a Spark application is submitted, the driver is always the first pod up for allocation. The most connected node is Node 1, resulting in the driver program pod being allocated to it.

Following this, we observe that among the nodes connected to Node 1, Nodes 2 and 4 have the highest number of neighbors. Consequently, subsequent pods will be assigned to Node 2 and Node 4. Therefore, the application will run on the fully-connected nodes 1, 2, and 4, causing no disruptions in its life cycle.

Algorithm 1 implements pod selection in a sequential fashion, similar to the `kube-scheduler` to simplify the integration effort with Kubernetes.

V. Implementation

We implement CASPR in 180 lines of Go [11], basing it on the open source implementation of the Kubernetes native scheduler [27]. We implement the connectivity monitoring overlay in 441 lines of C++ code.

Connectivity monitoring overlay. We implemented the daemons using C++. We run a connectivity daemon on each cluster node. A configuration file lists the IP addresses of all cluster nodes allowing daemons to heartbeat their peers. The heartbeats are performed over UDP, with a default heartbeat interval of 200 ms. Each daemon writes its connectivity matrix to a node-local file.

Scheduler. We implement the CASPR scheduler by making minimal changes to the native Kubernetes scheduler. We use Kubernetes v1.20.7 for our implementation. In particular we re-implement the `SelectHost()` function, a core function within the Kubernetes scheduler.

In our re-implementation of `SelectHost()`, called `SelectConnectedHost()`, we open and read the connectivity file created by the local connectivity daemon running on the same node as the scheduler. We map the information from this file to the node names Kubernetes stores to create the 2-dimensional connectivity matrix used for scheduling.

VI. Evaluation

Within this section we evaluate the performance of CASPR by comparing it to the native Kubernetes scheduler (`kube-scheduler`). We evaluate its fault tolerance capabilities, overheads and application performance under each partial network partition category outlined in Section III. We ran each experiment 10 times, reporting their averages and standard deviations. Although CASPR is capable of handling multiple concurrent network partitions, we insert a single network partition at any given time within our experiments to simplify our analysis.

Testbed. We conduct our experiments on a 5 node cluster at Cloudlab's [28] Utah data center. Because the behavior described in Section III is independent of cluster size, we use a small cluster to simplify deployment and debugging.

We use c6525-25g nodes [29] each of which had 16 AMD EPYC cores, 125GB RAM, and a 10 Gbps network connection. One of the nodes is always used to host the scheduler, while the other 4 are used as worker nodes. We use a separate set of nodes to host HDFS, to prevent the network partitions in our experiments affecting data placement.

Workloads. We use the WordCount workload outlined in Section III for our evaluation. We also include TeraSort, a workload that comes bundled with Spark [1]. TeraSort sorts a large amount of data via the MapReduce [2] paradigm and involves large network transfers during its shuffle phase.

A. Fault Tolerance

We evaluate CASPR's fault tolerance under the four partial partition scenarios we present in Section III. Table I shows the results of our evaluation, comparing CASPR with the other schedulers. Our evaluation shows that CASPR avoids the negative impacts of the previous schedulers (i.e. applications running using CASPR never get indefinitely stuck or experience significant performance degradation).

In all pre-existing partition scenarios, CASPR completely eliminates the negative impacts they impose. This is because CASPR ensures that for a given application, all application pods are allocated on fully-connected nodes. During mid-application M-W and M-D partition scenarios, CASPR behaves similarly to the `kube-scheduler`, ensuring that the partition does not impact the application.

An application running with CASPR experiences a small performance degradation during mid-application W-W or D-W partitions. In these two scenarios, the partition disrupts data transfer between workers or prevents the driver accessing the worker's results. In both scenarios, CASPR selects a new node to run the pod. This newly selected node is fully connected to the all other nodes hosting the application's pods.

The small performance degradation occurs due to the single re-execution of the task on this new node. Section VI-C shows that this degradation is minor compared to the `kube-scheduler` in the same scenarios.

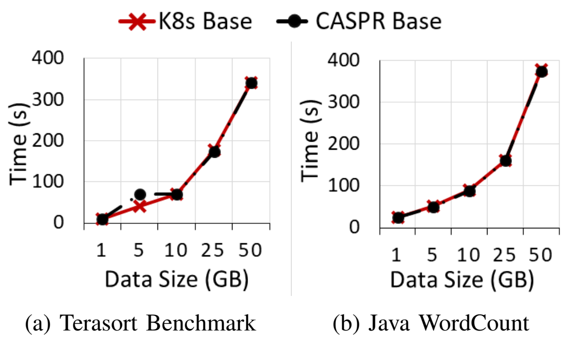


Fig. 3: Overhead evaluation in a partition-free scenario.

B. Overhead Evaluation

To measure the overhead CASPR imposes, we measure application execution time under a partition-free scenario when using CASPR and `kube-scheduler`. We ran WordCount and TeraSort for this experiment, varying input data sizes from 1 GB to 50 GB. The standard deviations of all runs were less than 6 %.

Figures 3a and 3b plot the application execution time against input data size, when running Terasort and WordCount respectively. Our results show that in partition-free scenarios, application execution time with CASPR is comparable to that with `kube-scheduler`, within 2% on average.

C. Performance Under Partial Partitions

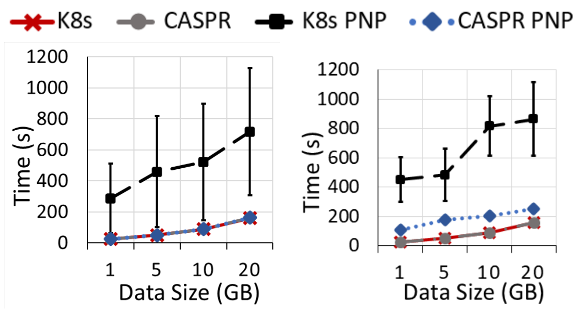
We compare application execution times with CASPR and `kube-scheduler` under the four partial partition categories outlined in Section III: W-W, D-W, M-W, and M-D. The partitions are inserted between nodes and not just target processes. We vary the time of partition injection, evaluating both pre-existing and mid-application partitioning scenarios within each category.

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:

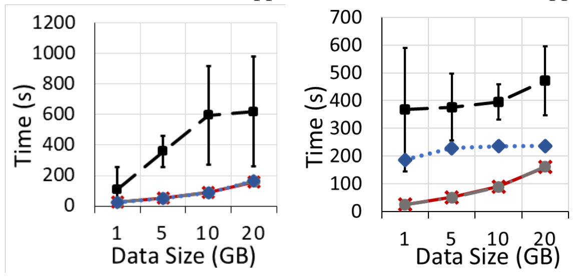
- *K8s Base* - `kube-scheduler` in a partition-free scenario.
- *CASPR Base* - CASPR in a partition-free scenario.
- *K8s PNP* - `kube-scheduler` experiencing the specified partial network partition.
- *CASPR PNP* - CASPR experiencing the specified partial network partition.

`W-W Partition.` Figures 4a and 4b show the performance in 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 *CASPR PNP* is faster than *K8s PNP* by $4 - 11\times$ on average. *K8s PNP* also causes a large variance in application execution time (Section III-D), as the error bars show.

A W-W partition disrupts the communication between two executor pods during the shuffle stage. Once `kube-scheduler` detects this problem, it tries to reallocate one of the pods. Unfortunately, the partition might also impact the new worker node chosen for the pod, causing

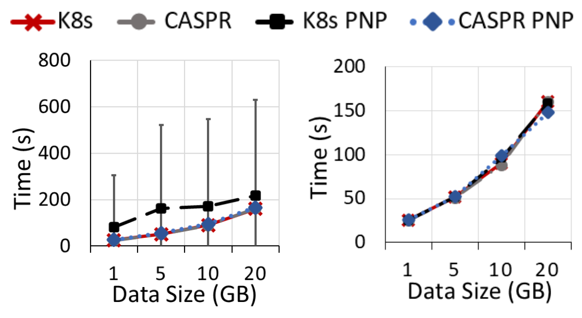


(a) W-W Partition Pre-App (b) W-W Partition Mid-App

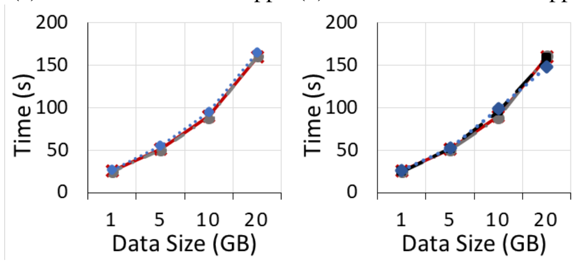


(c) D-W Partition Pre-App (d) D-W Partition Mid-App

Fig. 4: Average execution time of WordCount during W-W and D-W partitions. K8s Base, CASPR Base, and CASPR PNP have standard deviations less than 3%.



(a) M-W Partition Pre-App (b) M-W Partition Mid-App



(c) M-D Partition Pre-App (d) M-D Partition Mid-App

Fig. 5: Average execution time of WordCount under M-W and M-D partition. K8s Base, CASPR Base, and CASPR PNP have standard deviations less than 5%.

the communication to fail again and `kube-scheduler` to allocate a new pod onto a node. This process repeats until `kube-scheduler`, by luck, allocates a node that is fully connected to all application nodes.

Pre-existing. WordCount with CASPR experiences no performance degradation during pre-existing W-W partitions (Figure 4a). *CASPR PNP* has execution times similar to those of *CASPR Base* and *K8s Base*, because CASPR uses the information about cluster connectivity to schedule the application on a fully connected subset of nodes.

Mid-application. During mid-application partitions (Figure 4b), application execution time with *CASPR PNP* is 3.5× faster than *K8s PNP* on average and only 1.75× slower than that of *K8s Base*. When CASPR detects a problem between two executor pods, it allocates a new worker for one of them. It selects a worker that is fully connected to the application's active pods and the application resumes execution. Unlike `kube-scheduler`, CASPR restarts tasks only once.

`D-W Partition.` Figures 4c and 4d show the performance of CASPR and `kube-scheduler` under D-W partitions. *K8s PNP* suffers a significant performance degradation during both pre-existing (Figure 4c) and mid-application partitions (Figure 4d), with CASPR being 4-7× faster on average.

Pre-existing. With pre-existing partitions, `kube-scheduler` 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 fewer resources, leading to performance degradation. *CASPR PNP* in this case performs identically to partition-free scenarios, as CASPR 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 a new executor pod after a small delay depending on the driver heartbeat timeout. `kube-scheduler` can potentially allocate the new pod onto the same or a different partitioned node, causing the application to fail again as outlined above. With CASPR however, when the driver requests a new pod, CASPR will create a pod and allocate it to a node that is fully connected to the previous nodes. Therefore, CASPR will only repeat the failed task once. In this scenario CASPR is 2× faster than *K8s PNP*. The execution time difference between *CASPR PNP* and *K8s Base* in this case arises solely due to the detection delay described above.

`M-W Partition.` Figures 5a and 5b show the performance under M-W partial partitions.

Pre-existing. `kube-scheduler` suffers some performance degradation during pre-existing partitions. This is because, as explained in section III-D, it may assign pods to worker nodes that are unable to contact the scheduler, and the executor pods will fail to start. CASPR on the other hand will not allocate a pod onto a worker node that is partitioned from it, resulting in application performance very similar to that in partition-free scenarios.

Mid-application. During M-W mid-application partitions, all systems achieve similar performance. After an application starts, the driver program does not contact the schedulers (unless there is a pod failure) and thus is not impacted by the partial partition.

`M-D Partition.` Figures 5c and 5d show the performance in the M-D partial partition scenario.

Pre-existing. During pre-existing M-D partitions, the application does not run at all with `kube-scheduler` because the driver pod is allocated to a node that cannot communicate with the scheduler, causing the driver pod to never start. CASPR is able to run and execute successfully with execution times comparable to that partition-free scenarios as CASPR allocates the driver pod onto a node that is connected to itself.

Mid-application. With M-D mid-application partitions, all systems achieve similar performance. After an application starts, the driver program does not contact `kube-scheduler` barring pod failures, and thus is not affected by the partial partition.

Variance in Application Performance with *K8s PNP*. It is important to note the large variance in *K8s PNP* results. For instance, under W-W and D-W partitions (Figure 4), WordCount's execution time with *K8s PNP* ranges from 1 – 17 minutes with 5GB of data. This is due to the unpredictable number of reallocations `kube-scheduler` makes until it allocates the impacted executor pod on a node fully connected with the other active application pods (Section III-D). CASPR avoids this problem by using connectivity information to select a fully connected node.

VII. Related Work

Failure Studies. NEAT [6] and NIFTY [7, 8] present a study on the impact of network partitions, including partial partitions, on a diverse set of distributed systems. 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. 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%). While these studies motivated our work, they do not focus specifically on the impact of partial network partitions in scheduling and resource management systems.

In a large body of previous work, researchers have analyzed failures in distributed systems. A subset of these efforts focused on specific component failures such as physical [30] and virtual machines [31], network devices [14, 15], software bugs [32], storage systems [33, 34], and job failures [35, 36, 37]. Another set characterized a broader set of failures, but only for specific domains of systems and services, such as HPC [38, 39, 40], IaaS clouds [41], data-mining services [42], hosting services [43, 44], and data-intensive systems [32, 35, 45, 46].

Majumdar et al. [47] theoretically analyze the space for faulty executions with complete network partitioning faults. They discuss the extreme size of the test space and how effective it is to perform random testing if tests isolate a specific node, place a leader in a minority, and test with a random order of short sequences of operations. Bailis et al. [48] 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, because it is much easier to plan for partition failures ahead of time than to make changes to a complex system in a production environment.

We complement these efforts by studying the impact of partial partitions on scheduling for data analytics frameworks. Unlike previous efforts that relied solely on failure reports to study failures, we followed an empirical approach. We experimentally studied the impact of partial partitions on the state-of-the-art schedulers and designed a new scheduling approach using the insights from our study.

Fault Tolerant Scheduling. Although there are many scheduling systems for data analytics workloads [1, 49, 50], they do not tolerate network partitions, partial or complete. Sparrow [49] and Falcon [50] expose task failures to the application, allowing them to be handled in application specific format. Falcon uses a programmable switch to accelerate scheduling decisions, further complicating their fault tolerance model.

Elzeki et al. [51] review the many scheduling algorithms that can be applied in cloud computing, and how different characteristics such as the task arrival rate, task execution cost on each resource, and communication costs come into play when deciding which algorithm is best. Keivani et al. [52] 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 which account for these parameters.

Bala and Chana [53] 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 in scheduling.

VIII. Conclusion

We conducted an in-depth empirical study of the impact of partial network partitioning on modern resource management systems for data analytics frameworks. Our study shows that these systems 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 propose a new connectivity-aware scheduling technique (CASPR) that can tolerate partial partitions. CASPR incorporates cluster connectivity information in the process of scheduling. Our evaluation shows that CASPR can tolerate all partial network partitioning scenarios and imposes negligible overhead.

Our study highlights the importance of considering network failures and partial network partitions when designing and implementing scheduling techniques in modern distributed systems. By incorporating partition-resilience into their design, we can significantly increase their reliability and performance. The source code for CASPR is publicly available on GitHub [11].

References

[1] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. 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)*, pp. 15–28, 2012.

[2] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, (San Francisco, CA), pp. 137–150, 2004.

[3] A. Floratou, A. Agrawal, B. Graham, S. Rao, and K. Ramasamy, "Dhalion: Self-regulating stream processing in heron," *Proc. VLDB Endow.*, vol. 10, p. 1825–1836, aug 2017.

[4] V. Kalavri, J. Liagouris, M. Hoffmann, D. Dimitrova, M. Forshaw, and T. 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)*, (Carlsbad, CA), pp. 783–798, USENIX Association, Oct. 2018.

[5] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, pp. 74–80, 2013.

[6] A. Alquraan, H. Takruri, M. Alfatafta, and S. 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, (USA), p. 51–68, USENIX Association, 2018.

[7] M. Alfatafta, B. Alkhatib, A. Alquraan, and S. Al-Kiswany, "Toward a generic fault tolerance technique for partial network partitioning," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 351–368, USENIX Association, Nov. 2020.

[8] B. Alkhatib, S. Udayashankar, S. Qunaibi, A. Alquraan, M. Alfatafta, W. Al-Manasrah, A. Depoutovitch, and S. Al-Kiswany, "Partial network partitioning," *ACM Trans. Comput. Syst.*, dec 2022. Just Accepted.

[9] "Kubernetes Documentation." https://kubernetes.io/docs/home/, 2023. [Online; accessed 24-Mar-2023].

[10] R. Ignazio, *Mesos fundamentals*, p. 58–62. Manning, 2018.

[11] WASL, "Caspr github repository." https://github.com/UWASL/CASPR.

[12] R. Potharaju and N. 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), ACM, 2013.

[13] R. Govindan, I. Minei, M. Kallahalla, B. Koley, and A. Vahdat, "Evolve or die: High-availability design principles drawn from google's network infrastructure," 2016.

[14] P. Gill, N. Jain, and N. Nagappan, "Understanding network failures in data centers: Measurement, analysis, and implications," *SIGCOMM Comput. Commun. Rev.*, vol. 41, p. 350–361, aug 2011.

[15] D. Turner, K. Levchenko, A. C. Snoeren, and S. Savage, "California fault lines," *Proceedings of the ACM SIGCOMM 2010 conference*, 2010.

[16] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat, "B4: Experience with a globally-deployed software defined wan," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, (New York, NY, USA), p. 3–14, Association for Computing Machinery, 2013.

[17] T. Mills, "Bnx2 cards intermittantly going offline."

[18] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *SIGACT News*, vol. 33, p. 51–59, jun 2002.

[19] JIRA, "Mapreduce-4819: Am can rerun job after reporting final job status to the client." https://issues.apache.org/jira/browse/MAPREDUCE-4819.

[20] "Mapreduce ticket 4832." https://issues.apache.org/jira/browse/MAPREDUCE-4832. Accessed: April 2023.

[21] "Mesos-1529: Handle a network partition between master and slave." https://issues.apache.org/jira/browse/MESOS-1529. Accessed: April 2023.

[22] "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.

[23] "Flink architecture." https://nightlies.apache.org/flink/flink-docs-release-1.17/docs/concepts/flink-architecture/. Accessed: April 2023.

[24] "Apache storm documentation." https://storm.apache.org/releases/2.4.0/index.html/. Accessed: April 2023.

[25] "Hdfs architecture." https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html. Accessed: April 2023.

[26] J.-G. Perrin and R. Thomas, *Spark in action*. Manning Publications Co., 2020.

[27] Kubernetes, "Kubernetes/kubernetes: Production-grade container scheduling and management."

[28] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra, "The design and operation of CloudLab," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, (Renton, WA), pp. 1–14, USENIX Association, July 2019.

[29] T. C. Team, "Cloudlab documentation." https://docs.cloudlab.us/hardware.html. Accessed: April 2023.

[30] K. V. Vishwanath and N. Nagappan, "Characterizing cloud computing hardware reliability," SoCC '10, (New York, NY, USA), p. 193–204, Association for Computing Machinery, 2010.

[31] R. Birke, I. Giurgiu, L. Y. Chen, D. Wiesmann, and T. 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*, pp. 1–12, 2014.

[32] H. S. Gunawi, M. Hao, T. Leesatapornwongsa, and et. al, "What bugs live in the cloud? a study of 3000+ issues in cloud systems," in *ACM Symposium on Cloud Computing*, SOCC, 2014.

[33] D. Ford, F. Labelle, F. I. Popovici, and et. al, "Availability in globally distributed storage systems," in *USENIX Conference on Operating Systems Design and Implementation*, OSDI, 2010.

[34] W. Jiang, C. Hu, Y. Zhou, and A. Kanevsky, "Are disks the dominant contributor for storage failures? a comprehensive study of storage subsystem failure characteristics," *ACM Trans. Storage*, vol. 4, nov 2008.

[35] S. Li, H. Zhou, H. Lin, T. Xiao, H. Lin, W. Lin, and T. Xie, "A characteristic study on failures of production distributed data-parallel programs," in *2013 35th International Conference on Software Engineering (ICSE)*, pp. 963–972, 2013.

[36] X. Chen, C.-D. Lu, and K. Pattabiraman, "Failure analysis of jobs in compute clouds: A google cluster case study," in *IEEE International Symposium on Software Reliability Engineering*, pp. 167–177, 2014.

[37] P. Garraghan, P. Townend, and J. Xu, "An empirical failure-analysis of a large-scale cloud computing environment," in *IEEE International Symposium on High-Assurance Systems Engineering*, 2014.

[38] N. El-Sayed and B. 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)*, pp. 1–12, 2013.

[39] Y. Liang, Y. Zhang, A. Sivasubramaniam, M. Jette, and R. Sahoo, "Bluegene/l failure analysis and prediction models," in *International Conference on Dependable Systems and Networks (DSN)*, 2006.

[40] B. Schroeder and G. A. Gibson, "A large-scale study of failures in high-performance computing systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 4, pp. 337–350, 2010.

[41] T. Benson, S. Sahu, A. Akella, and A. Shaikh, "A first look at problems in the cloud," HotCloud'10, (USA), p. 15, USENIX Association, 2010.

[42] H. Zhou, J.-G. Lou, H. Zhang, H. Lin, H. Lin, and T. Qin, "An empirical study on quality issues of production big data platform," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2, pp. 17–26, 2015.

[43] D. Oppenheimer, A. Ganapathi, and D. 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), USENIX Association, Mar. 2003.

[44] H. S. Gunawi, M. Hao, R. O. Suminto, A. Laksono, A. D. Satria, J. Adityatama, and K. J. Eliazar, "Why does the cloud stop computing? lessons from hundreds of service outages," SoCC '16, (New York, NY, USA), p. 1–16, Association for Computing Machinery, 2016.

[45] A. Rabkin and R. H. Katz, "How hadoop clusters break," *IEEE Software*, vol. 30, no. 4, pp. 88–94, 2013.

[46] D. Yuan, Y. Luo, X. Zhuang, and et. al, "Simple testing can prevent most critical failures: An analysis of production failures in distributed Data-Intensive systems," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 249–265, Oct. 2014.

[47] R. Majumdar and F. Niksic, "Why is random testing effective for partition tolerance bugs?," *Proc. ACM Program. Lang.*, vol. 2, dec 2017.

[48] P. Bailis and K. Kingsbury, "The network is reliable: An informal survey of real-world communications failures," *Queue*, vol. 12, pp. 20–32, 2014.

[49] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: Distributed, low latency scheduling," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, (New York, NY, USA), p. 69–84, Association for Computing Machinery, 2013.

[50] I. Kettaneh, S. Udayashankar, A. Abdel-hadi, R. Grosman, and S. Al-Kiswany, "Falcon: Low latency, network-accelerated scheduling," in *Proceedings of the 3rd P4 Workshop in Europe*, EuroP4'20, (New York, NY, USA), p. 7–12, Association for Computing Machinery, 2020.

[51] O. Elzeki, M. Rashad, and M. Abu Elsoud, "Overview of scheduling tasks in distributed computing systems," *International Journal of Soft Computing and Engineering*, vol. 2, pp. 470–475, 01 2012.

[52] A. Keivani and J.-R. Tapamo, "Task scheduling in cloud computing: A review," in *2019 International Conference on Advances in Big Data, Computing and Data Communication Systems (icABCD)*, pp. 1–6, 2019.

[53] A. Bala and I. Chana, "Fault tolerance-challenges, techniques and implementation in cloud computing," *International Journal of Computer Science Issues*, vol. 9, 01 2012.