2016 IEEE International Conference on Cloud Engineering

Awan: Locality-aware Resource Manager for Geo-distributed Data-intensive Applications

Albert Jonathan, Abhishek Chandra, and Jon Weissman
Department of Computer Science and Engineering
University of Minnesota - Twin Cities
Minneapolis, Minnesota, USA
{albert, chandra, jon}@cs.umn.edu

***Abstract*—Today, many organizations need to operate on data that is distributed around the globe. This is inevitable due to the nature of data that is generated in different locations such as video feeds from distributed cameras, log files from distributed servers, and many others. Although centralized cloud platforms have been widely used for data-intensive applications, such systems are not suitable for processing geo-distributed data due to high data transfer overheads. An alternative approach is to use an Edge Cloud which reduces the network cost of transferring data by distributing its computations globally. While the Edge Cloud is attractive for geo-distributed data-intensive applications, extending existing cluster computing frameworks to a wide-area environment must account for locality. We propose `Awan` : a new locality-aware resource manager for geo-distributed data-intensive applications. `Awan` allows resource sharing between multiple computing frameworks while enabling high locality scheduling within each framework. Our experiments with the Nebula Edge Cloud on PlanetLab show that `Awan` achieves up to a 28% increase in locality scheduling which reduces the average job turnaround time by approximately 18% compared to existing cluster management mechanisms.**

Keywords*—*Geo-distributed; Cloud Computing; Resource Management; Scheduling; Edge Cloud; Data Intensive

I. Introduction

Today, many organizations deploy their applications and services around the globe for different organizational purposes and also rely on data generated in a geo-distributed manner. For example, a Content Delivery Network (CDN) has a number of servers distributed globally to deliver content to the end-clients with a low latency, and in turn collects user and performance logs at these locations. Such data collection can lead to many data analytic problems, such as client-activity log analysis, finding anomalies or interesting patterns in videos or images, and finding errors or security threats in servers. Similar geo-distributed data analysis is required for many applications in other domains such as social networking, scientific computing, and e-commerce.

Although centralized cloud platforms such as Amazon AWS [1] and Microsoft Azure [2] are popular platforms for data analytics, these centralized systems are not well suited for processing geo-distributed data. Using a centralized cloud platform for such applications requires data to be sent into a centralized location which will incur high network cost. Thus reducing data transfer cost in a wide area system is essential to the overall system performance. Due to the limitations of the centralized cloud architecture, an alternative approach is to use an *Edge Cloud* [3]–[5] that provides compute nodes closer to the edge. Such Edge Clouds can effectively exploit data locality to improve the performance of data analytics in geo-distributed environments.

Data-intensive applications are diverse in terms of their characteristics and requirements thus require different programming models to process the data efficiently. This has led to the emergence of a number of distributed computing frameworks such as Hadoop [6], Dryad [7], Pregel [8], and others [9], [10]. Although these computing frameworks were originally designed for processing applications in a centralized cluster environment, researchers have also looked at adapting and optimizing them in a geo-distributed environment to efficiently process geo-distributed data [11]–[13]. We believe that the growth of geo-distributed data and its applications will trigger more computing frameworks to be developed or adapted for a geo-distributed system. Thus, there is a requirement to share a limited number of resources across multiple computing frameworks in a geo-distributed environment. In general, sharing resources provides hardware cost benefits to users and also enables data consolidation which reduces the cost of replicating data across different locations. Although Resource sharing across multiple frameworks has been studied in a centralized cluster environment [14]–[16], these existing cluster management mechanisms do not scale well to an environment where the nodes are widely distributed, especially for data-intensive applications. In particular, these mechanisms lack the support for locality that is critical to achieving good performance in geo-distributed settings.

In this paper we introduce `Awan`[1], a new resource manager to share computing resources across multiple frameworks in an Edge Cloud environment. The main goal of `Awan` is to provide a general resource management mechanism that enables each computing framework's jobs to be scheduled with high locality, which is crucial in a geo-distributed environment. `Awan` achieves this goal by implementing a *resource lease* abstraction to allocate resources to individual *Framework Schedulers*. A lease provides a guarantee on the duration for which resources will be held by a scheduler. `Awan` provides this lease information to other *Framework Schedulers*, thus enabling them to make better scheduling decisions by considering the future availability of desirable local resources. In addition, we also propose a locality-based priority scheduling algorithm for intra-framework scheduling of jobs.

[1] Awan is an Indonesian word meaning "Cloud".

978-1-5090-1961-8/16 $31.00 © 2016 IEEE
DOI 10.1109/IC2E.2016.15

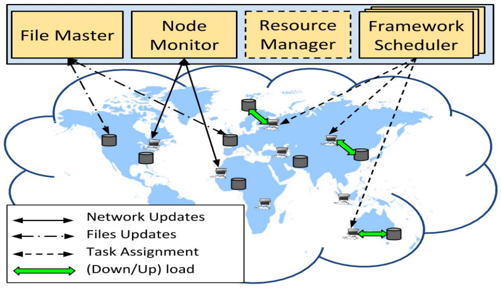


Fig. 1: Edge Cloud

Our experiments with the Nebula Edge Cloud [3] on PlanetLab [17] show that Awan outperforms the existing cluster management systems for data-intensive applications in a geo-distributed environment by increasing the locality scheduling that can be achieved by up to 28%. This locality improvement results in a better performance by reducing the average job turnaround time by approximately 18%. The use of the locality-based priority scheduling algorithm further improves the locality which results in a decrease in the average job turnaround time by an additional 13%.

II. Problem Context

We first introduce the system environment and the application model, and define some general terms that we use throughout this paper.

A. System Model

In this paper, we consider an Edge Cloud environment (see Figure 1): it consists of storage and compute nodes that are widely distributed around the globe. The nodes in this environment are on the edge of the network, and hence interconnected via a wide-area network. Both the storage and compute nodes are shared by multiple applications. A node in our environment may perform as a compute node, storage node or both, and multiple nodes may be physically running on the same machine. Throughout this paper, we will consider a *task* as the smallest granularity of application scheduling and a *compute node* as the smallest granularity of a computing resource that can be assigned to a task.

We assume that the Edge Cloud consists of following resource management components that are typical of many cloud environments:

File Master: All files that are stored in the system are managed by a *File Master*. The *File Master* is responsible for 1) maintaining file metadata, 2) handling file replication, and 3) determining which storage nodes are responsible for storing a specific file and its replicas. We will use the term file and data interchangeably throughout this paper.

Node Monitor: The health of each node needs to be monitored periodically to handle failures. We refer to the part of the system that performs a health-monitoring service as a *Node Monitor*. The *Node Monitor* is also responsible for monitoring the network bandwidth (up-link and down-link) between nodes. This bandwidth information is used to define the locality between nodes which will be used by the *File Master* for data placement decisions and by the *Framework Scheduler* to schedule tasks locally. A compute node is considered local to a storage node if they share the same machine or the network bandwidth between them is higher than a predetermined value.

Framework Scheduler: A *Framework Scheduler*[2] implements task scheduling logic for a specific computing framework (such as MapReduce [18] or MPI), and maintains the statistics of each task execution. As an example, for a data-intensive application executing on widely-distributed data, the *Framework Scheduler* would attempt to schedule its tasks with high locality because network bandwidth can be a dominant factor in the running time. Locality can be achieved by scheduling a task on a compute node that is closest[3] to the data location. We refer to this scheduling technique as *locality scheduling*.

Resource Manager: Since multiple users may want to run a variety of applications belonging to different frameworks, the system should allow its resources to be shared by multiple *Framework Schedulers*. The main goal of a *Resource Manager* is to provide a resource sharing service among *Framework Schedulers*. The *Resource Manager* periodically communicates with the *Node Monitor* to get information about online and offline nodes, and is also responsible for keeping track of the status of online nodes (e.g., available or busy, i.e., executing a task). Note that a system may not have a *Resource Manager*, in which case each *Framework Scheduler* will be able to allocate any resources directly. Such systems, however, introduce new challenges in coordinating the resource sharing policies across multiple *Framework Schedulers*.

B. Application Model

Our focus in this paper is on data-intensive applications where data locality is critical to achieving reduced running time. The applications in our environment operate on multiple data-sets that are geographically distributed and one or more frameworks may access the same data-sets. The application that is submitted by a user needs to specify which computing framework and which input data-sets are going to be used. For example, a user may submit a *Word Count* application to run on log files located on multiple servers that are distributed around the world using a Hadoop MapReduce [6] framework to find the number of errors that have occurred in a specific time period.

In general, an application, A, may consist of one or more *jobs*: $A = \{J_0, ..., J_{n-1}\}$ and a job J_i can be further broken down into multiple tasks: $J_i = \{T_{i,0}, ..., T_{i,n-1}\}$ where $n > 0$ and $i \geq 0$. Typically, a job is broken down into tasks based on the number of its input files. Each *Framework Scheduler* will select a job from its queue based on a job's priority and it schedules the tasks on a per-job basis. A task is the smallest granularity that is assigned to a compute node by a *Framework Scheduler* to run in parallel. It implements the programming logic to be executed on an input file and its inter-dependencies, if any, with other tasks. When a node receives a request, the

[2] Throughout this paper, we will use the term scheduler and Framework Scheduler interchangeably.

[3] Closeness is measured in term of the network bandwidth between two nodes, unless explicitly specified as a geographic distance.

node will first download the file, process it, and store the result to one or more storage nodes. For example, in a MapReduce application consisting of a Map and a Reduce job, the inputs of each Reduce task depends on the results of the Map tasks. Thus a Reduce task can only be run once all of the Map tasks that produce its inputs are completed.

III. Awan : A Shared Resource Manager

There have been several resource management mechanisms proposed for sharing resources among multiple *Framework Schedulers* in centralized cluster [14]–[16]. However, a key challenge in an Edge Cloud environment is the wide-area setting, thus there is a critical need for locality in scheduling data-intensive applications. We first begin by identifying some of the limitations of applying existing centralized resource management mechanisms in an Edge Cloud setting, and then present Awan: a resource manager that we have designed specifically for this environment.

A. *Limitations of Centralized Cluster Resource Managers*

At one extreme, one possible approach for sharing resources across multiple frameworks is to use a *Monolithic Scheduler*: a global scheduler that performs task allocation for every framework. Such a scheduler would implement a general scheduling logic that can be used by variety of frameworks. Although a *Monolithic Scheduler* simplifies resource sharing, the general scheduler in this model is difficult to extend with new framework-specific policies and optimizations. Hence, it is difficult to support a wide variety of applications or extend its support to new frameworks.

In order to have an optimized task allocation policy for each framework, researchers have implemented multiple *Framework Schedulers*, each of which is optimized for a specific computing framework. However, sharing limited number of resources across multiple *Framework Schedulers* introduces new challenges such as resource partitioning, concurrency control issues, prioritizing certain jobs on different frameworks, and many others. One approach for sharing resources would be to statically partition the resources in advance and give each *Framework Scheduler* a predetermined subset of resources. A static partitioning approach, however, leads to an external fragmentation problem and low resource utilization. Setting the size of each partition may also be difficult if each framework has a dynamic workload that changes over time. Moreover, a static partitioning approach is not suitable for a geo-distributed setting since it limits the locality scheduling for each *Framework Scheduler*.

Dynamic resource partitioning solves the external fragmentation problem by enabling the size of each partition to change dynamically depending on the workload. A *two-level architecture* uses a dynamic partitioning model. It consists of a single *Resource Manager* that performs as an abstraction layer between the resources and the *Framework Schedulers*. The *Framework Schedulers* are independent of each other and they interface with the *Resource Manager* in order to acquire resources. Mesos [14] is a resource management system that uses a two-level architecture. In Mesos, all *Framework Schedulers* acquire resources from the *Resource Manager* using a *resource offer* mechanism. In this model, a *Framework Scheduler* would request available resources from the *Resource Manager* when there is a job that needs to be scheduled. Upon receiving this request from the scheduler, the *Resource Manager* would offer available resources that satisfy the scheduler-defined constraints as closely as possible. In return, the *Framework Schedulers* may either accept or reject the offer if the offered resources do not adequately satisfy the requirements.

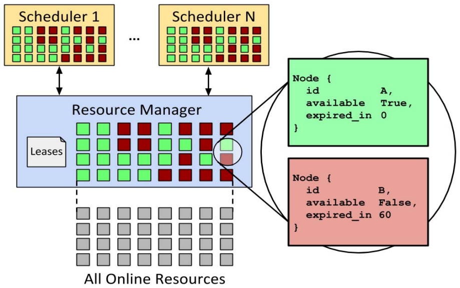
The resource offer mechanism uses a pessimistic concurrency control, meaning that the resources that are currently offered to one *Framework Scheduler* will not be offered to another *Framework Scheduler* at the same time. This ensures there is no conflict between *Framework Schedulers* in allocating resources. A clear drawback of the pessimistic approach is that only one scheduler can acquire a particular set of resources at a time. Thus, the remaining schedulers may not be able to allocate desirable local resources if they are already busy, or may have to wait indefinitely for the *Resource Manager* to offer these resources to them. An alternative approach would have the *Resource Manager* perform global resource allocation for all requests it receives from the *Framework Schedulers*, similar to the approach used in YARN [15]. This, however, makes the two-level architecture effectively monolithic since the resource allocation is determined by a single global resource allocator.

A *shared-state architecture* that was introduced in Google Omega [16] has no *Resource Manager*, thus each *Framework Scheduler* has a direct access to execute any of its tasks on any of the available resources. In this architecture, the state of all resources are shared by all schedulers that can schedule their tasks in parallel using an optimistic concurrency control. This mechanism gives all of the schedulers knowledge about all online resources and their states (available or busy). This knowledge, however, is only used to avoid a scheduler trying to acquire busy resources. While the shared-state architecture is useful in a cooperative environment, using it in an Edge Cloud with limited number of resources may lead to some problems such as fairness and starvation since different *Framework Schedulers* may be competitive and/or try to hoard resources. Furthermore, since a shared-state architecture does not have a coordinator in managing the resources given to the schedulers, implementing global policies may be difficult. Every time a global policy is added, it may require some changes in every *Framework Scheduler* to account for the new policy.

B. *Awan Resource Manager*

To address the limitations of the existing cluster resource managers in an Edge Cloud environment, we propose a new resource manager called Awan. The goal of Awan is to provide a scalable resource sharing mechanism in a geo-distributed system with high locality scheduling which is needed for data-intensive applications. Awan combines the desirable features of the two-level architecture with those of the shared-state architecture, while providing explicit support for locality-aware scheduling. Figure 2 shows the two-level architecture of Awan. We incorporate the shared-state mechanism by sharing the states of all the resources to every *Framework Scheduler*. In our system, however, the states of the resources are shared by the *Resource Manager* and not directly by the *Framework Schedulers*.

The *Resource Manager* in Awan provides the states of all resources instead of only the resources that are available.



```
Node {
  id          A,
  available   True,
  expired_in  0
}
```

```
Node {
  id          B,
  available   False,
  expired_in  60
}
```

Fig. 2: Awan's Two-level Architecture

The *Framework Schedulers* acquire and schedule available resources using a *resource lease* mechanism with an *optimistic concurrency control* as in Omega. However, the optimistic concurrency control in acquiring resources is handled by the *Resource Manager*. To handle conflicting resources, the resources will be acquired in an atomic manner. Any global policies that need to be obeyed by the *Framework Schedulers* are enforced by the *Resource Manager*. For example, to implement fair-sharing between schedulers, the *Resource Manager* may limit the number of resources that can be acquired by a scheduler based on the number of active frameworks.

C. Resource Lease

In a geo-distributed environment, the resource offer mechanism that is used by Mesos suffers from the potential lack of locality in task scheduling. The main reason is due to the limited knowledge of resource availability that is provided by the *Resource Manager*, since a resource offer mechanism only offers currently available resources. At a glance, offering only the available resources seems reasonable because tasks *can* only be scheduled on available resources. A busy resource, however, may actually provide better data locality for a task than the currently available resource *when* it becomes available in the future. Mesos handles this issue by incorporating a delay scheduling [19] for short running tasks which makes a scheduler delay scheduling a job if its task cannot be launched locally. However, delay scheduling without any knowledge of the future availability of the resources, esp. if most of the tasks are long-running, may worsen the performance by introducing unnecessary waiting time. Instead, it would be desirable for a scheduler to wait (or not) on busy resources depending on the expected waiting time: if the waiting time is too long, the scheduler may prefer to schedule its tasks non-locally. Thus, sharing the future availability information of busy resources can help a scheduler make a better scheduling decision.

A lease in the *resource lease* mechanism has a *lease expiration time* associated with it which provides a guarantee that the acquired resources will be held by a scheduler no longer than the lease time (with a possibility of some grace period). After the lease time expiry, the *Resource Manager* will make the resource available to the other schedulers. The sharing of this lease time information enables schedulers to estimate their waiting time for desired busy resources, leading to improved scheduling decisions.

When a *Framework Scheduler* tries to acquire some available resources $R_0, R_1, \ldots, R_{n-1}$, the scheduler sends a *lease request* $< L_0, L_1, \ldots, L_{n-1} >$ on each of the resources to the *Resource Manager*. Here, L_i is the lease time on resource R_i. If the *Resource Manager* agrees on the request and the resources are available, the resources will be granted to the scheduler and the state of each of the resources will be moved to a busy state. The request may also contain an *atomic_request* flag specifying all resources must be acquired *atomically*. If the *atomic_request* flag is set to *false*, the leases on available resources will be granted and the scheduler will be notified if any of the leases failed. If the flag is set to *true*, the request will be granted iff *all* the leases can be satisfied, i.e., all resources in the request are available. Failure in leasing may happen because of the optimistic concurrency control used in our resource management mechanism, where multiple schedulers may try to acquire overlapping sets of available resources at the same time. We provide such an atomic request option because some computing frameworks (e.g., MPI) require all resources to be available to start the execution, while others (e.g., MapReduce) can start a job partially and add more resources later.

The *Resource Manager* maintains all the leases from *Framework Schedulers* and they are shared to all *Framework Scheduler*. Since each lease contains information about the future availability of a resource, a scheduler may decide whether to schedule its tasks on the available resources or wait for the busy resources. If a scheduler decides to wait on one or more resources, the scheduler should be able to dynamically change its scheduling decision over time. This is useful for a few reasons. First, the network performance between two nodes is constantly changing over time, especially in a system that is connected via WAN. Thus a scheduler may change its scheduling decision if the network performance drastically changes. Second, failure is common in a distributed system, thus a scheduler should be able to adapt its scheduling decision if the resource it wants, and is waiting for, has failed. Third, the lease estimation that is provided by other schedulers may not be perfectly accurate (in practice it is unlikely to have a 100% prediction accuracy). If the resource becomes available sooner than the estimated time, the scheduler will be able to acquire the node right away. We will analyze the problem of lease estimation in the next section. Lastly, in an optimistic concurrency control, multiple schedulers may wait for the same resource. Since only one scheduler is able to acquire the resource, the other schedulers should be able to reschedule the task on a different resource. In the latter problem, a *waiting list* can be added to each resource and this information can also be shared to every scheduler.

In addition, we use a two-level architecture instead of the single-level shared-state architecture to allow global policies to be applied to every scheduler easily. Some policies that are incorporated in our implementation are: 1) the capability of rejecting a lease request for an unreasonable long time and 2) terminating a process that takes longer than lease time. If fairness between *Framework Schedulers* or users is the main priority, fair sharing may be applied by limiting the number of resources that can be leased using max-min fair sharing.

D. Lease Estimation and Enforcement

When a scheduler tries to select a node for running a task, it needs to estimate the time needed to complete the task. The lease time in our implementation is estimated by combining the network cost and the statistical history of similar tasks' running time. However, having a perfect accuracy in estimating the lease time is not possible for many reasons. A wide area network that is shared publicly causes fluctuations in network bandwidth, and networking problems such as packet loss may also influence the data transfer time. Estimating the network bandwidth between nodes interconnected via WAN is a challenging problem that trade-offs between the estimation time and accuracy level [20], [21]. In our system implementation, the network cost estimation is best-effort and it is estimated based on the bandwidth estimation that is shared by the *Node Monitor* from the statistics of recent data transfers between nodes. Although, the network cost is the dominant factor for wide-area data intensive applications, we also consider the other computation factors into the lease calculation to have a better lease estimation accuracy. These factors are estimated as a per-node compute performance history of running similar tasks, which are maintained by each *Framework Scheduler*. In general, the lease time of a compute node, L_a, is estimated as follow:

$$L_a = \frac{datasize}{B_{b,a}} + \bar{C_k} + \delta \tag{1}$$

L_a is the estimated time needed to complete a task using a compute node a. $B_{b,a}$ is the current bandwidth that can be utilized between a storage node b and a compute node a. $\bar{C_k}$ is the average compute time of the newest k^{th} records for processing similar tasks using a compute node a. The maximum number of records, k, is used to avoid including obsolete records into the calculation. For simplicity, we consider the size of a file that is processed by a task for a framework to be the same. Thus, a large file will be partitioned into multiple files. The similarity of the task can be categorized based on the computing framework, the application of the task, and the data size. However, since we limit the maximum data size of a task (bigger data size will result in multiple tasks) and the task is not compute-intensive, we can consider the task to have running time similar to its historic value. If there are no similar tasks in the history of the compute node a, the $\bar{C_k}$ will be estimated from the performance of running a similar task on a different node. If none of the nodes have ever computed the task, the scheduler will lease the node for a predefined amount of time. This might cause a large inaccuracy in running a task for the first time. A small slack factor, δ, is added to the estimation to avoid an overly optimistic prediction.

Both underestimation and overestimation of lease time can lead to problems. If the lease time is underestimated, meaning that the time needed to complete a task is longer than the lease time, the task would not be completed before the lease expiration time. If the *Resource Manager* kills the task running on the expired node, this will result in wasted resources and increase the turnaround time for the task's parent job. The overall system utilization and performance will deteriorate significantly with a high number of lease underestimations. On the other hand, if the lease time for a node is overestimated, fewer schedulers may wait for this node to become available, and may instead schedule their tasks on less desirable non-local nodes. With a high number of lease overestimations, most schedulers will ignore the busy nodes and schedule their tasks on the available nodes, effectively reducing to a resource offer-based mechanism. This might result in non-local execution of tasks, thus decreasing the overall system performance.

The *Resource Manager* can also use different policies to handle expired leases. The simplest approach is to terminate the running task upon lease expiry and set the node to be available to other schedulers. However, terminating a process that has not finished on an expired lease requires the task to be rescheduled on a different node, and will result in wasted resources if the task is almost finished. A better approach is to give some *grace period* for the node to clean up or finish. If the task is a long-running task and the progress of the process is far from completion, the state of the task could be saved and any temporary results should be stored to a storage node such that the task can be continued by another node instead of restarting the whole task. The *Resource Manager* should carefully determine what is the appropriate grace period. If the grace period is too low, it is likely to result in large number of terminated tasks. On the other hand, if the grace period is too high, it may lead to much higher waiting times for other schedulers waiting for the node to become available.

IV. Locality-based Priority Scheduling

Our discussion so far has focused on how to maintain high locality while sharing resources between multiple frameworks. In this section, we further look at how to improve locality in scheduling multiple jobs for a specific framework. A notable technique to improve locality in scheduling is by using a delay scheduling algorithm [19], which would skip a job that is at the head of the job-queue if any of its tasks cannot be scheduled locally. To avoid starvation, the scheduler can limit the number of times a job is skipped. Once the number of skips reaches a predetermined threshold, the job will be scheduled even if its tasks cannot be scheduled locally. This technique is feasible if most of the tasks are short-running-tasks where a short delay is often sufficient to have a higher locality. Introducing a delay in scheduling long running tasks, however, works well only if a scheduler has a complete knowledge of the status of all nodes (including busy nodes). If a scheduler is only aware of the nodes that are available (on which it can launch its tasks without any delay), delaying tasks may incur unnecessary waiting time since it is possible that a task cannot be scheduled locally in any of the nodes. In our resource management mechanism, the waiting time for the nodes can be obtained from the lease information that is shared by the *Resource Manager*. If the waiting time is too long, the scheduler should be able to schedule the task to a different resource right away.

We generalize the delay scheduling algorithm by introducing a *minimum locality level constraint*. Instead of immediately skipping a job if any of its tasks cannot be scheduled locally, we compare the locality level of the job with the minimum locality level constraint. The locality level of a job, *Loc*, is defined as the fraction of tasks that the scheduler can launch locally, which can be computed as follow:

$$Loc = \frac{\sum_{t \epsilon T} \begin{cases} 1 & \text{if } B \geq B_{min} \\ 0 & otherwise \end{cases}}{|T|} \tag{2}$$

t is a task from a set of tasks, T, that are going to be scheduled, $|T|$ is the cardinality of T, B is the bandwidth used for running

t and B_{min} is the minimum bandwidth that determines the locality. A job can only be scheduled if $Loc \geq Loc_{min}$, where Loc_{min} is the minimum locality that is set by the scheduler. A minimum locality level of 0 means that the job will be scheduled regardless of the number of tasks that can be scheduled locally. On the other hand, a minimum locality level of 1 means that a job can be scheduled only if all of the tasks can be scheduled locally (effectively similar to the delay scheduling algorithm).

A minimum locality level constraint of 0 might result in locality scheduling for a low-intensity workload since each scheduler tries to schedule its tasks locally and can find such resources available. On the other hand, an overly high minimum locality level constraint may lead to a high waiting time due to the restriction on scheduling. In practice, adjusting the minimum locality level constraint to some value i, where $0 < i \leq 1$, may increase the number of tasks that can be scheduled locally since it could prioritize jobs that have higher locality. The maximum number of skips should also be set carefully since a higher number of skips would result in a higher waiting time. In summary, the minimum locality level should be adjusted to the scheduler's workload and the average number of tasks that can be scheduled locally from the statistical history.

V. Experimental Evaluation

A. Experimental Setup

We used the Nebula Edge Cloud [3] for our experiment platform. Nebula is a geo-distributed system which utilizes volunteer nodes for both its computation and storage nodes. We modified the monolithic scheduler in Nebula to the two-level Awan Resource Manager that shares the resources using a leasing mechanism. Awan , however, could be expanded into any Edge Cloud that provides the system model in section II.

Since the focus of this paper is on exploring the impact of wide-area distribution on application performance, we did not focus on handling the unreliability of volunteer nodes in Nebula. When there was a straggler or a node was down during task execution, the scheduler would be notified and it simply rescheduled the task to another node. To protect and isolate the compute nodes from malicious codes as part of the applications, Nebula uses a Google Chrome Web browser-based Native Client (NaCl) sandbox [22]. Any code execution is carried out inside the sandbox to ensure the safety of the nodes. We deployed Nebula on PlanetLab [17] Europe (PLE) nodes located across more than 15 countries for both the compute and storage nodes. We used 40 compute nodes and 32 storage nodes for most of the experiments. The nodes in PlanetLab are heterogeneous in both their computation power and bandwidth (varying from less than 1Mbps to more than 10Mbps). In our experiment, we considered a node to be a *local node* if it provided a bandwidth connection higher than 8Mbps. Most of the storage nodes that we deployed had at least one local compute node, but not all. The centralized services such as *Node Monitor*, *File Master*, *Resource Manager*, and *Framework Schedulers* were hosted on a dedicated machine with an Intel Xeon CPU E5-2609 and 16GB of memory.

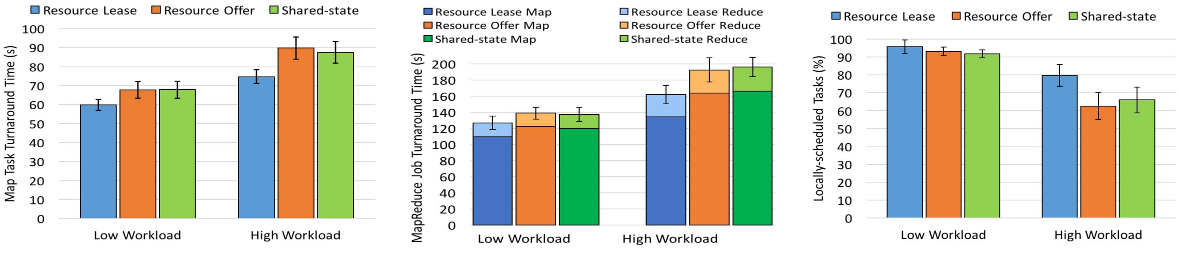
To emulate the resource sharing among multiple *Framework Schedulers*, we implemented a *MapReduce* (MR) scheduler, a *First-Come-First-Serve* (FCFS) scheduler, and a *Random* scheduler. All of the schedulers are running on top of the *Resource Managers* (except for the shared-state architecture). We used the logic of the geo-distributed MapReduce mechanism described in [3]. Both the MR and FCFS would schedule their tasks locally as far as possible whereas the Random scheduler always selected nodes randomly. The MR scheduler would consider delay scheduling its tasks if waiting for a local node was a better option. The FCFS scheduler, on the other hand, would never consider waiting for local nodes and skipping a job with low locality. So, if a local node was not available, the FCFS scheduler would assign the task to a randomly selected node. In our experiments, we omitted the results from the Random scheduler since the scheduler did not account for the locality benefit from our approach. The purpose of including a Random scheduler was to add a higher workload in the system by randomly acquiring some resources in the background.

In our experiments, all of the schedulers would schedule a set of homogeneous MapReduce Word Count jobs with different data-sets (varied from 256MB to 512MB per job). We also assumed that the data had already been distributed randomly across the storage nodes as 16MB chunks. Thus, each task in every job would run on a maximum of 16MB data. The critical path of each of the MapReduce jobs in our experiments was in the Map job which: download the data, process the data, and store the result back to one or more storage nodes. The computation of the Reduce tasks in our experiments were not critical to the overall MapReduce performance since the inputs of the Reduce tasks (the output of the Map tasks) were much smaller compared amount of data computed by the Mappers. Hence, the benefit of locality can be better seen in computing the Map tasks. We ran a background process that randomly posted a job from 12 homogeneous jobs (each operates on a different data-set) using a *Poisson Process* with different inter-arrival rates. We used a job inter-arrival rate of 100 seconds in a "Low Workload" and 50 seconds in a "High Workload" which resulted in 1 to 2 concurrent jobs and 2 to 4 concurrent jobs respectively. On average, a task could be completed in about 40 to 60 seconds if it was run on a local node and could take more than 100 seconds if it was run on a non-local node. In our leasing mechanism, we gave a 20 second grace period to every lease upon expiration since the misestimation for most lease is within 20 seconds (unless explicitly specified).

B. Benefit of Awan Resource Manager

1) Resource Management Comparison: To compare the Awan *Resource Manager* to other existing resource management mechanisms, we implemented mechanisms similar to those used in the resource offer and the shared-state mechanisms within the Nebula infrastructure. The *Resource Manager* in the resource offer mechanism offered all available resources (omitting any busy nodes) whenever a scheduler requested compute resources. On the other hand, every scheduler in the shared-state mechanism had a visibility to all of the online resources, including those that were busy. However, the shared information only indicated whether a node was busy or available, and did not have any lease time information.

In this experiment, we used 8 homogeneous MapReduce



(a) Map Task Turnaround Time (b) MapReduce Job Turnaround Time (c) Average Locality Achieved

Fig. 3: Comparison of Different Resource Management Mechanisms

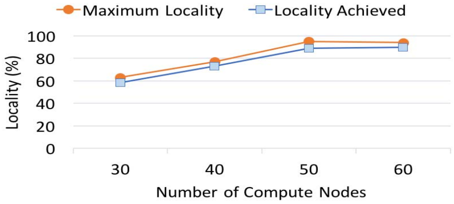


Fig. 4: Locality On Different Number Of Compute Nodes

Word Count jobs where each of them operated on a 256MB data-set. We compared the result in a ”Low Workload” and ”High Workload” with 5 iterations. Figure 3a and 3b show the comparison on the average task turnaround time and job turnaround time respectively. In the ”Low Workload”, all of the resource management mechanisms performed similarly because most of the jobs could be scheduled locally. However, the performance differences can be seen in the ”High Workload”. In the ”High Workload”, our resource management mechanism resulted in 18% reduction in the average Map task execution time which leads to the 18% reduction of the average MapReduce job turnaround time (the turnaround time is determined by the last completed task). These improvements resulted from the higher number of local tasks that was scheduled (28% higher locality), as can be seen in Figure 3c. The main reason for the higher locality was because of the knowledge of the availability time for each node that was shared to all schedulers via lease information. Thus, the scheduler in our resource management mechanism would wait for local nodes to achieve locality scheduling.

The schedulers in the resource offer mechanism did not know about the existence of the busy nodes because the *Resource Manager* offered only the available nodes. In this case, a scheduler would only try to schedule its tasks locally on a subset of the nodes that had been offered. This led to a lower locality scheduling that could be achieved. The states that were shared in the shared state mechanism only provided information whether a node was available or busy. This information was only used by the scheduler to avoid scheduling tasks on a busy node. Waiting for busy nodes that can be used for locality scheduling in a shared-state architecture is not appropriate since a scheduler does not know the availability time for the node. Thus, a scheduler in this architecture would only try to schedule its tasks locally on the nodes that were available, which is similar to the resource offer mechanism.

2) Scalability: We also explored the locality that can be achieved with a growing number of nodes using our lease mechanism. In this experiment, we changed each job to run on 32 tasks instead of 16 tasks with a total data size of 512MB. We increased the number of storage nodes to 60 and varied the number of compute nodes from 30 to 60. The jobs were posted using a Poisson Process with 8 jobs and an inter-arrival rate of 60 seconds.

Figure 4 shows the locality that was achieved vs. the maximum locality that could be achieved over different number of compute nodes. The maximum locality that could be achieved was low when the number of compute nodes was low because the data was randomly distributed throughout a much higher number of storage nodes. However, the locality that was achieved was close to the maximum locality that could be achieved (about 5% difference on average) for each configuration of compute nodes. Thus, the schedulers could schedule their tasks achieving close to the best possible locality regardless of the number of compute nodes.

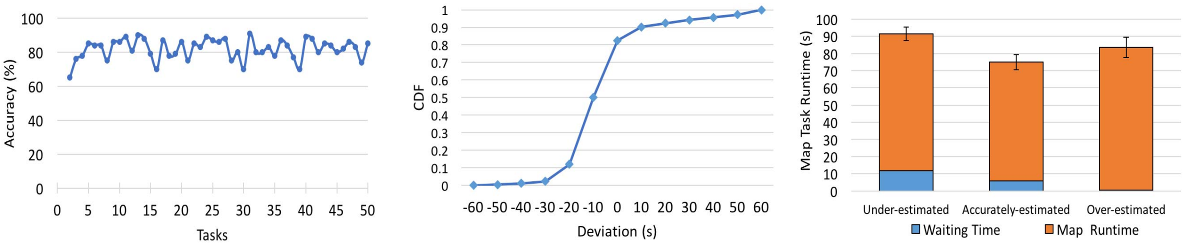
C. Lease Estimation and Enforcement

1) Lease Estimation: Since our resource management depends critically on the lease estimation that is provided by each *Framework Scheduler*, the estimation should reflect the actual time needed to complete a task. Achieving a perfect lease time accuracy may not be possible due to the dynamic nature of a wide area network and a machine’s workload. However, having a close estimation is possible by exploiting knowledge of the task’s running time from its statistical history and the monitored network performance between nodes. In this experiment, we deployed 8 MapReduce jobs, each of which ran on 256MB data-sets. The jobs were posted using a Poisson Process with a rate of 100 seconds.

Figure 5a and 5b show the accuracy and the CDF of the lease estimation in our system respectively. The accuracy is defined as:

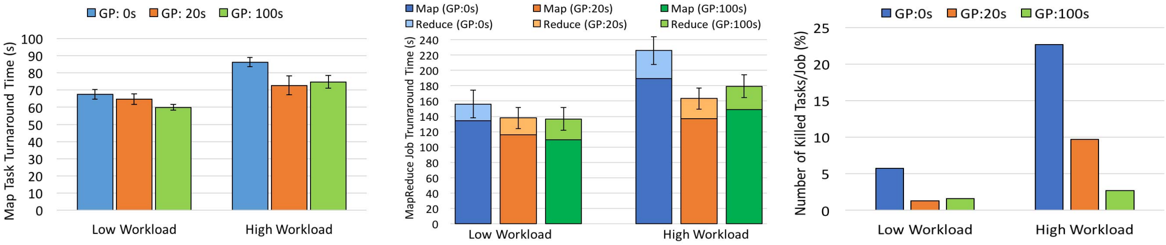
$$accuracy = 1 - \frac{|runtime - prediction|}{runtime} \tag{3}$$

On average, the lease estimation we used results in about 82% accuracy. There were some points in Figure 5a where the accuracy dropped below 80%. The main reason was because of new jobs that were posted or stragglers that appeared during the experiments. The latter can be explained in Figure 5b. If there were tasks whose information was not available, a scheduler would estimate the task’s running time using a predefined conservative value (100 seconds lease time was



(a) Lease Estimation Accuracy (b) Lease Estimation CDF (c) Misestimation On Task Turnaround Time

Fig. 5: Lease estimation



(a) Map Task Turnaround Time (b) MapReduce Job Turnaround Time (c) Killed Tasks/Job

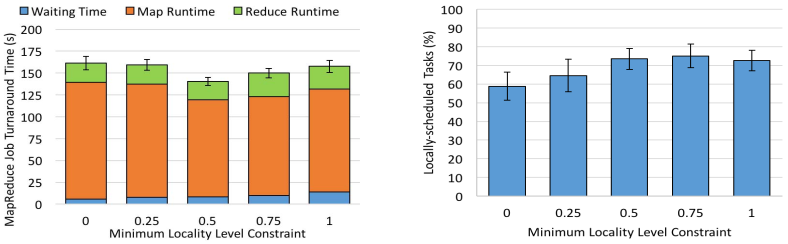
Fig. 6: Impact Of Different Grace Periods

used in the experiment). Thus the lease estimation time resulted in low accuracy whenever new jobs arrived. When similar jobs were posted later on, the scheduler could predict the task running time better from its statistical records. However, the accuracy still fluctuated over time even if similar jobs were posted due to a slight inconsistency in the network bandwidth. We can also see there were some stragglers that appeared in a deviation greater than 50 seconds which causes the reduction in prediction accuracy.

We also looked at the impact of lease time misestimation on performance. We observed the problems of overestimation and underestimation on the lease estimation by varying the δ value in the lease estimation equation (Equation 1). The δ value was set to -30 seconds for the *underestimated* and +100 seconds for the *overestimated* cases, respectively. We selected -30 seconds for the underestimated value since it was the approximate time difference from running local and non-local task which result in unnecessary waiting time. The +100 seconds for the overestimated value was chosen simply to make schedulers give up the local task execution. We ran the experiments with 10 iterations each of which used a Poisson Process with a rate of 50 seconds. In this experiment, we gave an expired lease a very long grace period to prevent the vast majority of tasks from being killed (except for the occasional straggler). Figure 5c shows the effect of misestimation on the average task turnaround time. The *underestimated* result shows the highest average waiting time or delay (about 12 seconds), whereas the *overestimated* time resulted in less than 1 second average task waiting time. Most of the jobs in the *underestimated* results were delayed by the schedulers because the schedulers preferred to wait for local nodes with short waiting time. However, the information that was shared by the *Resource Manager* to the schedulers was not accurate, thus most of the delays were unnecessary. On the other hand, a highly *overestimated* lease time resulted in a very low average task waiting time. Since every scheduler leased nodes with a highly overestimated time, other schedulers rarely waited for local nodes and decided to schedule their tasks non-locally. This was the main reason for the increase in the running time. Thus, having an accurate estimation for the lease time is desirable, since a misestimation can increase the average task running time by up to 20%.

2) Handling Expired Leases: We compared 3 different techniques to handle expired leases. The first one was to immediately terminate the process running on expired node (no grace period was given). The second one was to give a small grace period (20 seconds was used in the experiment) to an expired lease. The schedulers that had been waiting for the node were notified and guaranteed that the node would be available within the grace period. If the task could not be completed even after the grace period ended, the task would be killed. The last one was to give a very high grace period (100 seconds) to the expired lease. We ran the experiments with 10 iterations using a Poisson Process for each workload.

Figure 6 compares the system's performance with different grace periods. Figure 6a and 6b show that a strict lease with no grace periods resulted in the worst performance for the average map tasks running time and MapReduce job turnaround time respectively. The main reason to the low performance was due to the high number of tasks that were killed, as shown in Figure 6c. As mentioned previously, having a perfect estimation is not possible and there were a few tasks that took slightly longer than the estimated time. In a strict lease, these tasks were restarted even if they only had a few seconds left. Figure 6c shows that when a small grace period was given, the number of tasks that were killed reduced significantly. This resulted in a much better system performance compared to the strict lease (12% and about 28% improvements on average job turnaround time in a "Low Workload" and "High Workload"



(a) MapReduce Job Turnaround Time

(b) Locality Achieved

Fig. 7: Comparison Of Different Minimum Locality Levels

respectively). A lease with too high a grace period, however, results in worse performance compared to the 20 seconds grace period although the number of tasks that were killed was much lower. The tasks that were killed in the 100 seconds grace period were basically stragglers. In a ”High Workload”, there were multiple schedulers that waited for nodes for locality scheduling. Giving a very high grace period to an expired lease made most of the schedulers give up waiting on the nodes and to schedule the tasks non-locally, thus leading to a slight increase in the average job turnaround time by 8%. However, the difference is not significant since most of the tasks in our experiments completed before the lease expired.

D. Locality-based Priority Scheduling

In this experiment, we evaluated the benefit of prioritizing higher locality jobs using a *minimum locality level*. This technique is used to reduce the number of tasks that are scheduled on non-local nodes, which may prevent locality scheduling for the other tasks. A job that could not achieve the minimum locality level constraint will be skipped for no more than a predefined amount of time, set to 10 seconds. In this experiment, we varied the minimum locality level from 0 to 1. The locality level 0 means that a job can be scheduled regardless of the number of local tasks that it tries to schedule. On the other hand, if the minimum locality level is set to 1, a job can only be scheduled if all of the tasks are scheduled to run locally, which is effectively similar to the delay scheduling algorithm [19].

In this experiment, the resources were shared among the schedulers using the resource-lease mechanism. We only show the results for a ”High Workload” because for a ”Low Workload” the schedulers would already schedule their tasks with high locality. The jobs were posted using a Poisson Process with a rate of 50 seconds and we posted 2 jobs at a time instead of 1 job to allow multiple jobs to reside in the job queue.

Figure 7a shows the average job turnaround time for different minimum locality levels. As we can see from the figure, the average job turnaround time decreased by 13% when the minimum locality level increased from 0 to 0.5. However, the average job turnaround time increased when the minimum locality level was set too high even if the number of tasks that were locally scheduled was about the same as shown in Figure 7a. The increase of the average job turnaround time was caused by the delay that was enforced by the minimum locality constraint. Figure 7b shows the locality level achieved with different minimum locality levels. If a scheduler enforced a high level of minimum locality level that must be satisfied in order to schedule, more jobs would be skipped due to the limited number of local tasks. Thus, the minimum locality level should be adjusted dynamically depending on the number of resources and the workload.

VI. Related Work

Geo-distributed Systems: There have been projects that utilize geographically distributed computing resources (similar ot the grid computing) [11]–[13], [23]–[25]. However, most of them focus on extending or adapting a specific computing framework to a wide-area environment, but not considering resource sharing between multiple computing frameworks. There are also projects that utilize geographically distributed nodes for storages [26]–[30]. Our work utilizes both geo-distributed compute and storage nodes, but focuses more on the sharing the compute nodes. Moreover, most of these systems focus on inter-data-center environments, which is different from the wide-area Edge Cloud environment that we target in `Awan`.

Resource Management: Sharing resources across multiple frameworks in a cluster is not a new topic. There have been projects that observe the challenges of sharing resources between multiple schedulers [14]–[16], [31]. These mechanisms, however, are intended for a centralized cluster which requires modifications to scale to a wide-area environment. The architecture that we used is similar to Mesos. However, we use a leasing mechanism which is different from the resource-offer mechanism in Mesos. We incorporate the shared state mechanism that is used in Omega with further modification by sharing the future availability of every node to every scheduler. The technique that we used is similar to Apollo [32], which estimates the running time for each task and shares it with every scheduler. While Apollo and Omega focus on resource sharing in a centralized cluster as opposed to `Awan` which is designed for a geo-distributed system.

Framework Scheduling: Researchers have also considered optimizing individual framework schedulers in a cluster. Torque [33] is a batch scheduler for High Performance Computing cluster where data locality is not the main issue for such jobs. Delay scheduling [19] and Quincy [34] incorporate techniques to handle locality and fair sharing in a co-located cluster where the data is stored on the nodes that run the jobs. In delay scheduling, a scheduler would delay for a pre-defined short period of time if a job cannot be scheduled locally. This technique is sufficient if the task running time is short. If the task running time is long, which is common in a wide-area system, the waiting time should be measured to determine whether delay scheduling is necessary.

VII. Conclusion and Future Work

In this paper, we presented Awan: a resource management mechanism for data-intensive applications in a geo-distributed system. Our mechanism allows resource sharing across multiple Framework Schedulers while maintaining high locality for each framework. Since network cost is expensive in a widely distributed system, improving locality scheduling greatly reduces the average task turnaround time for data-intensive applications. We also looked at improving locality in job scheduling by prioritizing jobs with higher locality over lower locality jobs.

In future work, we plan to investigate how other global policies can be enforced in Awan without sacrificing locality. We will also explore a more diverse and heterogeneous set of resources which may include inter-cluster data-center nodes, to enable Awan to support resource management in multiple data centers. We also plan to include each resource's reliability in the process of selecting resources in Awan to enable the inclusion of more volatile volunteer nodes such as in Nebula.

References

[1] Amazon web services. [Online]. Available: http://aws.amazon.com

[2] Microsoft azure. [Online]. Available: https://azure.microsoft.com

[3] M. Ryden, K. Oh, A. Chandra, and J. Weissman, "Nebula: Distributed edge cloud for data intensive computing," in *Cloud Engineering (IC2E), 2014 IEEE International Conference on*. IEEE, 2014, pp. 57–66.

[4] S. Ibrahim, H. Jin, B. Cheng, H. Cao, S. Wu, and L. Qi, "Cloudlet: towards mapreduce implementation on virtual machines," in *Proceedings of the 18th ACM international symposium on High performance distributed computing*. ACM, 2009, pp. 65–66.

[5] D. P. Anderson, "Boinc: A system for public-resource computing and storage," in *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*. IEEE, 2004, pp. 4–10.

[6] Hadoop. [Online]. Available: https://hadoop.apache.org/

[7] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3. ACM, 2007, pp. 59–72.

[8] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 135–146.

[9] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, vol. 10, 2010, p. 10.

[10] B. He, M. Yang, Z. Guo, R. Chen, B. Su, W. Lin, and L. Zhou, "Comet: batched stream processing for data intensive distributed computing," in *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 2010, pp. 63–74.

[11] B. Heintz, C. Wang, A. Chandra, and J. Weissman, "Cross-phase optimization in mapreduce," in *Cloud Engineering (IC2E), 2013 IEEE International Conference on*. IEEE, 2013, pp. 338–347.

[12] A. Rabkin, M. Arye, S. Sen, V. S. Pai, and M. J. Freedman, "Aggregation and degradation in jetstream: Streaming analytics in the wide area," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, 2014, pp. 275–288.

[13] Q. Pu, G. Ananthanarayanan, P. Bodik, S. Kandula, A. Akella, P. Bahl, and I. Stoica, "Low latency geo-distributed data analytics," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. ACM, 2015, pp. 421–434.

[14] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center." in *NSDI*, vol. 11, 2011, pp. 22–22.

[15] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2013, p. 5.

[16] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: flexible, scalable schedulers for large compute clusters," in *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, 2013, pp. 351–364.

[17] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman, "Planetlab: an overlay testbed for broad-coverage services," *ACM SIGCOMM Computer Communication Review*, vol. 33, no. 3, pp. 3–12, 2003.

[18] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[19] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *Proceedings of the 5th European conference on Computer systems*. ACM, 2010, pp. 265–278.

[20] R. Wolski, "Dynamically forecasting network performance using the network weather service," *Cluster Computing*, vol. 1, no. 1, pp. 119–132, 1998.

[21] J. Kim, A. Chandra, and J. Weissman, "Open: Passive network performance estimation for data-intensive applications," *Dept. of CSE, Univ. of Minnesota, Tech. Rep*, pp. 08–041, 2008.

[22] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native client: A sandbox for portable, untrusted x86 native code," in *Security and Privacy, 2009 30th IEEE Symposium on*. IEEE, 2009, pp. 79–93.

[23] M. J. Litzkow, M. Livny, and M. W. Mutka, "Condor-a hunter of idle workstations," in *Distributed Computing Systems, 1988., 8th International Conference on*. IEEE, 1988, pp. 104–111.

[24] A. Vulimiri, C. Curino, B. Godfrey, K. Karanasos, and G. Varghese, "Wanalytics: Analytics for a geo-distributed data-intensive world," in *Conference on Innovative Data Systems Research (CIDR 2015)*, 2015.

[25] C.-C. Hung, L. Golubchik, and M. Yu, "Scheduling jobs across geo-distributed datacenters," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*. ACM, 2015, pp. 111–124.

[26] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild *et al.*, "Spanner: Googles globally distributed database," *ACM Transactions on Computer Systems (TOCS)*, vol. 31, no. 3, p. 8, 2013.

[27] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. V. Madhyastha, "Spanstore: Cost-effective geo-replicated storage spanning multiple cloud services," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 292–308.

[28] A. Bessani, R. Mendes, T. Oliveira, N. Neves, M. Correia, M. Pasin, and P. Verissimo, "Scfs: a shared cloud-backed file system," in *Proc. of the 2014 USENIX Annual Technical Conference*, 2014.

[29] K. Oh, A. Raghavan, A. Chandra, and J. Weissman, "Redefining data locality for cross-data center storage," *BigSystem 2015: 2nd International Workshop on Software-Defined Ecosystems*, 2015.

[30] S. Agarwal, J. Dunagan, N. Jain, S. Saroiu, A. Wolman, and H. Bhogan, "Volley: Automated data placement for geo-distributed cloud services." in *NSDI*, 2010, pp. 17–32.

[31] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 2015, p. 18.

[32] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou, "Apollo: scalable and coordinated scheduling for cloud-scale computing," in *Proc. USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, 2014.

[33] Torque resource manager. [Online]. Available: http://www.adaptivecomputing.com/products/open-source/torque

[34] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: fair scheduling for distributed computing clusters," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009, pp. 261–276.