Nebula: Distributed Edge Cloud for Data Intensive Computing

Albert Jonathan, *Student Member, IEEE*, Mathew Ryden, *Student Member, IEEE*, Kwangsung Oh, *Student Member, IEEE*, Abhishek Chandra, *Member, IEEE*, and Jon Weissman, *Senior Member, IEEE*

Abstract—Centralized cloud infrastructures have become the popular platforms for data-intensive computing today. However, they suffer from inefficient data mobility due to the centralization of cloud resources, and hence, are highly unsuited for geo-distributed data-intensive applications where the data may be spread at multiple geographical locations. In this paper, we present Nebula: a dispersed edge cloud infrastructure that explores the use of voluntary resources for both computation and data storage. We describe the lightweight Nebula architecture that enables distributed data-intensive computing through a number of optimization techniques including location-aware data and computation placement, replication, and recovery. We evaluate Nebula performance on an emulated volunteer platform that spans over 50 PlanetLab nodes distributed across Europe, and show how a common data-intensive computing framework, MapReduce, can be easily deployed and run on Nebula. We show Nebula MapReduce is robust to a wide array of failures and substantially outperforms other wide-area versions based on emulated existing systems.

Index Terms—Distributed Systems, cloud computing, edge cloud, data intensive computing

1 Introduction

Today, centralized data-centers or clouds have become popular platforms for data-intensive computing in the commercial, and increasingly, scientific domains. The appeal is clear: clouds such as Amazon AWS [1] and Microsoft Azure [2] offer large amounts of monetized co-located computation and storage services that are well suited for processing batch analytic applications. These centralized systems, however, are not well suited for many data-intensive applications that rely on data that are geographically distributed for a couple reasons. First, data upload may constitute a non-trivial portion of the execution time since the data is sent through a shared wide area network (WAN) which is known to have limited bandwidth, high latency, and costly. Second, data upload coupled with the high overhead in instantiating virtualized cloud resources, further limits the range of applications to those that are either batch-oriented or long-running services. In this paper, we present Nebula, a system that utilizes edge resources for both computation and data storage to address the two aspects addressed above. We advocate the use of volunteer resources but Nebula can also use monetized resources across CDNs or even ISPs that are equipped to offer computational services.

• *The authors are with the University of Minnesota Twin Cities, Computer Science & Engineering, Minneapolis, MN 55455. E-mail: {albert, ohkwang, chandra, jon}@cs.umn.edu, mathew.ryden@gmail.com.*

Manuscript received 7 Oct. 2016; revised 12 May 2017; accepted 16 June 2017. Date of publication 21 June 2017; date of current version 11 Oct. 2017. (Corresponding author: Albert Jonathan.)
Recommended for acceptance by Z. Chen.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2017.2717883

To give a flavor of why the edge may be suitable for data-intensive computing, consider the following examples. A user wishes to analyze a set of blogs located across the Internet to identify opinions about the recent election. Another user wants to analyze video feeds from geo-distributed cameras across a set of airports looking for suspicious activity. These examples have the following characteristics that make the edge attractive: each data can be processed independently in-situ (or close to the data location) and the data processing typically yields to significant data compression. In the first case, only a small subset of blogs is returned (any mention of politics) and in the latter case only portions of the videos that contain potentially suspicious activity are needed for further processing. These examples show that a huge amount of data is by nature generated in a dispersed manner and can benefit from filtering, pre-processing and aggregation. So, using the edge resources that provide in-situ processing can significantly reduce the amount of data that need to be sent to the final centralized location.

Nodes in edge cloud can constitute different levels of granularity ranging from regional data centers [3], [4], [5], [6], [7] as the coarsest granularity to cloudlets or small servers deployed in ISPs [8], [9] as the finest granularity. There is a clear trade-off between the two models. The regional data center model typically provides more storage capacity and computational resources within each location compared to the more dispersed model. However, the regional data center model will have a higher overhead to transfer data between the edges because they are relatively far away form each other compared to the more dispersed model. Data transfer between edges is very common in an edge cloud environment. For example, a system may replicate each data and distribute the copies into multiple storage nodes located in different regions of the globe to provide high data availability. Furthermore, many geo-distributed data analytic applications perform data aggregation on multiple input data sets that are located in different locations around the globe. The dispersed edge cloud model can significantly mitigate this overhead since the edges are closer to each other. Thus, the decision on which model to use highly depends on the application of interests. In this paper, we focus on data-intensive applications whose inputs are highly distributed. For such applications, the WAN bandwidth consumption typically becomes the bottleneck. Hence, we consider the more dispersed edge cloud model to minimize the network overhead.

Building a dispersed edge cloud system requires a fine grain distribution of resources. Using volunteer resources can be a very attractive option to build such system since many of them are by nature widely distributed. Furthermore, volunteer resources become more attractive with the provision of powerful multi-core home machines coupled with increasing amount of Internet connectivity bandwidth that is available today. However, the use of volunteer resources makes the system more challenging since the resources they provide are highly heterogeneous, they are typically less reliable, and unpredictable, i.e., a volunteer node may leave the system at anytime. Thus, a volunteer-based system needs to handle these challenges.

In this paper, we introduce a geo-distributed edge cloud system, called Nebula, that explores the use of volunteer resources to democratize in-situ data-intensive computing. Although we focus on the voluntary aspect of Nebula, the system and tehcniques that we propose are not limited to volunteer resources. Nebula allows dedicated resources in addition to volunteer resources to mitigate the reliability issues of volunteer-based systems. In contrast to existing volunteer platforms such as BOINC [10], which are designed for compute-intensive applications, and file-sharing systems such as BitTorrent [11], our system is designed to support *distributed data-intensive applications through a close interaction between compute and storage resources*.

Nebula implements a number of optimizations to enable efficient exploitation of edge resources for in-situ data-intensive computing, including location-aware data and computation placement, replication, and recovery. In this paper, we focus on the systems and implementation aspects of Nebula. We show how a common data-intensive computing framework, MapReduce [12], can be easily deployed, and run on Nebula. In addition, we also explore how the system handles external data upload and processing, and show that Nebula is robust to a range of failures of both hosted compute and storage nodes that may occur anywhere within the system. We evaluate Nebula's performance on an emulated volunteer platform that spans over 50 PlanetLab [13] nodes distributed across Europe and show that Nebula can greatly outperform two volunteer platforms that have been proposed in the literature, standard BOINC [10] and BOINC-MR [14] on our testbed through a number of locality-based optimization techniques and exhibit good scaling properties.

2 Nebula Overview

2.1 Design Goals

Nebula has been designed with the following goals in mind:

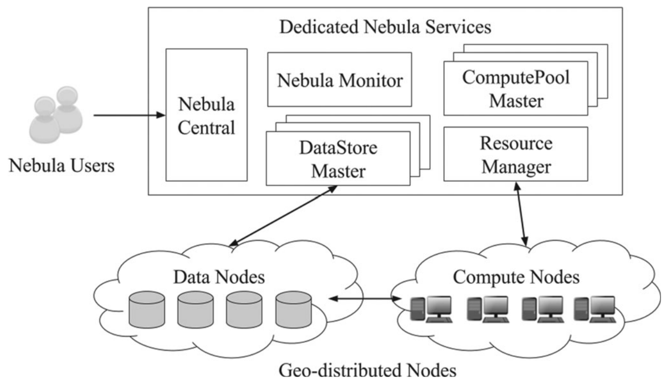


Fig. 1. Nebula system architecture.

- *Support for distributed data-intensive computing*: Unlike other volunteer computing frameworks such as BOINC that focus on compute-intensive applications, Nebula is designed to support data-intensive applications that require efficient movement and availability of large quantities of data to compute resources. As a result, Nebula must also support a scalable data storage platform. Further, Nebula is designed to support applications where data may originate in a geographically distributed manner, and is not necessarily preloaded to a central location.
- *Location-aware resource management*: To enable efficient execution of distributed data-intensive applications, Nebula must consider network bandwidth along with computational capabilities of resources in the volunteer platform. As a result, resource management decisions must optimize for computation time as well as data movement costs. In particular, compute resources may be selected based on their locality and proximity to their input data, while data may be staged closer to efficient computational resources.
- *Secure execution environment*: To ensure that volunteer nodes are completely safe and isolated from malicious code that might be executed as part of a Nebula-based application, volunteer nodes must be able to execute all user-injected application code within a protected sandbox.
- *Ease of use*: Nebula must be easy to use and manage both for users who execute their applications on the volunteer platform, as well as for volunteer nodes that donate their resources. In particular, users should be able to easily inject their application code and data into Nebula for execution. At the same time, volunteer nodes must be able to easily join Nebula and start volunteering with low overhead.
- *Fault Tolerance*: Nebula must ensure fault tolerant execution of applications in the presence of node churn and transient network failures that are common in a volunteer environment. Nebula must also ensure a high level of data availability of storing data in volunteer storage nodes[1].

2.2 Nebula System Architecture

Fig. 1 shows the Nebula system architecture. Nebula consists of nodes that donate their computation and/or storage

1. We will use the term storage node and data node interchangeably.

resources, along with a set of global and application-specific services that are hosted on dedicated, stable nodes. These resources and services together constitute five major components in Nebula (described in Section 3):

- *Nebula Central*: Nebula Central is the front-end for the Nebula ecosystem. It provides a simple, easy-to-use web-based portal that allows nodes to join the system, users to upload/download their data, application writers to inject applications into the system, and tools to manage and monitor application execution.
- *DataStore*: The DataStore is a simple per-application storage service that supports efficient and location-aware data storage in Nebula. Each DataStore consists of data nodes that store the actual data, and a DataStore Master that keeps track of the storage system metadata and makes data placement decisions.
- *ComputePool*: The ComputePool provides per-application computational resources through a set of compute nodes. Code execution on a compute node is carried out inside a Google Chrome web browser-based Native Client (NaCl) sandbox [15]. Compute nodes within a ComputePool are scheduled by a ComputePool Master that coordinates their execution. The compute nodes use the DataStore to access and retrieve data, and they are assigned tasks based on application-specific requirements and data location.
- *Nebula Monitor:* The Nebula Monitor does performance monitoring of nodes and network characteristics. This monitoring information consists of node computation speeds, memory and storage capacities, and network bandwidth, as well as health information such as node and link failures. This information is dynamically updated and is used by the DataStore and ComputePool Masters for data placement, scheduling and fault tolerance.
- *Resource Manager:* The Resource Manager provides support for sharing compute nodes among multiple ComputePools. This resource sharing mechanism is performed dynamically depending on factors such as the current availability of the resources, the number of concurrent applications, the reliability of the resources, and so on.

2.3 Nebula Applications

A Nebula application A may consist a number of jobs: $A = \{J_0, \ldots, J_n\}$, where $n \geq 0$. A job contains the code to carry out a specific computation. An application execution is referred to as an instantiation of the application. An application is typically associated with an input dataset, which consists of multiple data objects (files). The input dataset can be centrally located or geographically distributed across multiple locations. We explore cases where an application uses data that has already been stored into the Nebula storage system as well as those which use external data that has not been stored in Nebula to begin with. A job may also depend on other jobs, in which case the dependent job will not be executed until all of its predecessors are complete, and its input dataset may be specified as the output(s) of the predecessor job(s). Each job J_i consists of multiple tasks $\{T_{i,0}, \ldots, T_{i,m}\}$, where $m \geq 0$. These tasks are typically identical in structure and can be executed in parallel on multiple compute nodes.

TABLE 1
Nebula Central Inputs

Argument	Applicability	Function
title	Application	The human-readable name of the application instantiation.
namespace	Application	The namespace for input and output files.
source files	Application	A list of filename to use as source inputs for the jobs.
output file prefix	Application	The filename prefix to use for outputs of jobs.
job scheduler	Application	The scheduler to be used by the ComputePool Master.
metadata	Per-Job	Job information such as the whether it uses source files or depends on other jobs.
javascript	Per-Job	The javascript code to use when running a job for computation.
executables	Per-Job	The NaCl executables for the jobs.
DataStore parameters	Per-Job	The parameters for input data locations and parameters for jobs for output.
task parameters	Per-Job	Setting for a task including timeouts, minimum bandwidths, replication settings, and failure tolerances.

Each task is associated with the job executable code and a data partition of the input dataset, upon which the code is executed. In Nebula, the executable code is Native Client code (a set of `.nexe` or `.pexe` files) that can be executed in the Native Client sandbox on the compute nodes. The input data is identified by a set of filenames that refer to data already stored in Nebula. After this, the compute nodes retrieve data from the DataStore to execute the corresponding tasks.

3 Nebula System Components

3.1 Nebula Central

Nebula Central is the front-end for the Nebula ecosystem. It is the forward facing component of the system for managing applications and resources in the system. It exposes a web front-end portal where nodes can join to contribute compute and/or storage resources to Nebula, and users can upload their applications and initiate execution. Nebula Central uses a number of parameters (Table 1) to create an expanded internal representation of the entire application, the component jobs, and the set of tasks. It also instantiates an application-specific DataStore and ComputePool Master (to be discussed next) that handle the data placement and job execution for the application. It then provides the necessary application components to the DataStore and ComputePool Masters, such as the application executable code.

In Nebula, a volunteer node can join the system as a compute node or a data node or both by registering with Nebula Central. A data node needs to download a generic piece of code that can carry out data node functions such as storing/retrieving files, and talking to DataStore Masters and clients. On the other hand, a compute node only requires a Native Client-enabled Web browser to carry out its tasks, and does not need any additional Nebula-specific software. In the future, we envision that data node volunteers will also use a browser to download their infrastructure via signed Java applets or similar technology. Once a volunteer node joins the system, the node is then assigned to one or more DataStores and/or ComputePools, and interacts directly with their respective

Master nodes to carry out data storage and/or computation respectively on behalf of the corresponding applications.

3.2 DataStore

The DataStore is designed to provide a simple storage service to support data processing on Nebula. In a highly dispersed edge cloud model, nodes are typically interconnected by WAN that has limited bandwidth. Optimizing data transfer time is crucial to efficient data processing in this case. Each application owns a DataStore that it may configure to support desired performance, capacity, or reliability constraints. Data nodes that are part of a DataStore may be multiplexed across potentially other DataStores.

Data in a DataStore is stored and retrieved in unit of files. Files are organized using a combination of namespace and filename. For instance, different users may be assigned different namespaces, and a user may further partition their files (such as those belonging to different applications) into different namespaces. Files with the same filename but belonging to different namespaces are thus considered distinct. We support flat namespaces denoted by simple per-DataStore unique strings which can be used to partition files much like directory names. Files are considered immutable and file appends or edits are not supported. A DataStore consists of multiple data nodes that store the actual data, and a single DataStore Master that manages these nodes and keeps track of the storage system metadata.

3.2.1 *Data Node*

The data nodes donate storage space and store application data as files. A data node is implemented using a light-weight web server that stores and serves files. Nebula allows dedicated data nodes in addition to volunteer data nodes for a couple reasons. First, data that is stored in Nebula may have a privacy concern, thus the owner may only want to store the data in trusted dedicated nodes. Second, volunteer-based data nodes tend to be less reliable. Hence, users may choose to store critical data in dedicated nodes that provide higher reliability. For example, input data may be stored in highly reliable data nodes whereas the intermediate data can be stored in less-reliable data nodes which trades-off the reliability aspect for higher locality to the compute nodes.

In general, the data nodes support two basic operations that are exposed to clients to store and retrieve files:

- `store(filename, namespace, file)`: stores a file into the DataStore and update the DataStore Master.
- `retrieve(filename, namespace)`: retrieves the file specified by the namespace-filename combination.

DataStore clients, such as compute nodes that need to store or retrieve data to/from the DataStore, use a combination of these operations to get and store data. These operations can also be used to place data in a desired fashion before the computation even begins. All the intelligence is part of the DataStore Master and is transparent to the clients.

3.2.2 *DataStore Master*

The DataStore Master keeps track of several items: data nodes that are online, file metadata, file replication, and data placement policy. It serves as an entry point to the DataStore when writing to or reading data from it. The information it collects is used to make decisions regarding the placement and retrieval of data. The DataStore Master supports a set of operations to manage the data nodes and to carry out effective data placement decisions. Some of these operations are invoked by DataStore clients before storing or retrieving data to/from the DataStore. Other operations are invoked by the data nodes and are used to exchange metadata and health information:

- `get_nodes_to_store(option)`: returns an ordered list of data nodes. The order of the nodes depends on the option specified by the client such as the load of data nodes, closest nodes to the client, availability of compute nodes around the data node or random.
- `get_nodes_to_retrieve(filename, namespace)`: returns an ordered list of data nodes that stores a particular file that could be located on multiple nodes.
- `set(filename, namespace, nodeId, filesize)`: sets a new entry in the DataStore Master for the given file. This operation is invoked by a data node once the upload is completed successfully.
- `ping(nodeid, is_online)`: reports that a node is online. The data nodes periodically ping the DataStore Master to notify their status. Data nodes that do not check in after a period of time are marked as offline. The second argument also provides an option for a node to manually report itself as offline upon a graceful exit or shutdown.

These operations are implemented over an HTTP protocol, so parameters like clients IP address/location are implicitly available via the request headers sent by the client. Bandwidth and latency information are computed and tracked on the fly, whenever there is data transferred between nodes in the system. The effective bandwidth between nodes is tracked by dividing the total amount of data that is sent by the time it takes from sending the first data to the last acknowledgement received by the sender. The latency, on the other hand, is first estimated based on the distance between two nodes' coordinates, which is a reasonable estimation as shown in previous work [16], [17]. Once, there is a data exchange between nodes, the latency will be updated by averaging the currently monitored latency with the average time of receiving acknowledgement for each data exchange. Since the bandwidth information is not available for newly joined nodes, we insert a random position in the returned node list. This is to ensure that new nodes do not get starved, and bandwidth information can eventually be discovered for such nodes. In the future, we hope to employ a heuristic based on the node characteristics to order such nodes.

3.2.3 *Load Balancing and Locality Awareness*

To access data that has already been stored in the DataStore, the DataStore Master can provide better performance via load balancing and locality awareness by appropriately ordering the list of data nodes returned to the clients via the `get_nodes_to_retrieve` operation. To achieve a good load balancing, the list of nodes can be randomized or ordered by their load (or available storage capacity). By

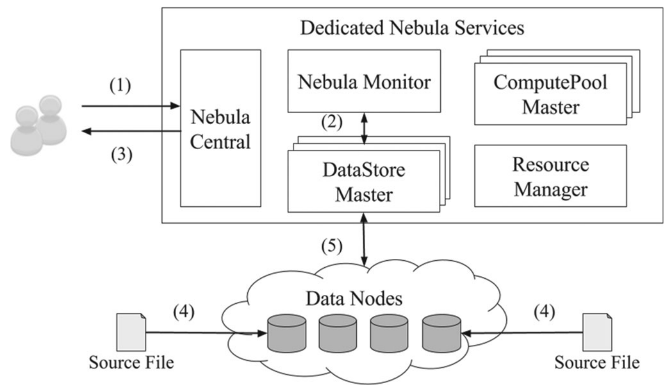


Fig. 2. Nebula upload mechanism.

ordering the nodes based on their load, new data will be stored in a data node that has the lowest load. This ordered list will be updated whenever a data node joins/leaves the system or there is an update on existing nodes' load information. Locality awareness, on the other hand, can be achieved by ordering the nodes based on their bandwidth or latency from clients. As an example, consider a case where a node is both a compute node and a data node. For requests originating from that particular compute node, the operations will, by default, place the local storage node at the head of the list.

The DataStore Master is aware of the health and location of all data nodes by periodically monitoring them using heartbeat/ping requests. This is useful since volunteer nodes in Nebula may leave at anytime. The location of a data node can be used to provide low latency support to the users by allowing them to store/access their data to/from data nodes that are closely located to the users. We estimate the latency of a user to a node based on their coordinates. This locality awareness is especially critical for low latency critical applications such as data caching in mobile cloud computing or fog computing [9].

Latency, however, may not always be a good metric to define locality for geo-distributed data-intensive applications since data transfer between nodes is typically constrained by the WAN bandwidth rather than the latency [18], [19], [20]. For each data node, the DataStore Master keeps track of its effective bandwidth to other nodes by maintaining a key-value map data structure where the key is a node ID and the value is the monitored bandwidth between the nodes. When a DataStore client, such as a ComputePool Master, wants to determine which node has the highest locality to a given data node, it will send a query to the DataStore Master and get the node-bandwidth mapping sorted by the bandwidth. Thus, the ComputePool Master can use this as a hint to schedule a task to a compute node with the highest bandwidth (highest locality) to mitigate the time required to transfer data from the data node to the compute node.

3.2.4 External Data Upload

In many scenarios, data may not already be stored in the DataStore, but may have to be uploaded into it from an external source (similar to data upload into a centralized cloud). Nebula supports a data upload mechanism by which an external file is directly transferred to a set of selected data nodes (returned by the `get_nodes_to_store` operation from the DataStore Master). This direct data transfer mechanism is beneficial for a number of reasons. First, it removes centralized components such as the Nebula Central and DataStore Master from the data path, thus avoiding centralized bottlenecks and increasing the scalability of data transfer. Second, it enables finding preferred network paths between the source and destination, and also enables parallel file transfer along multiple network paths. Nebula also provides flexibility to allow users to upload their files from remote servers by providing the URI of the source files. The DataStore Master selects a desired set of data nodes, and the selected data nodes then download the files from the provided URI. This mechanism also allows a user to upload multiple files from different location in parallel to geographically distributed data nodes.

Fig. 2 shows the uploading mechanism in Nebula. (1) A user will first access a web page from the Nebula Central that provides an interface for data uploading. (2) The Nebula Central will forward the query to the DataStore Master which will make a data placement decision. There are a few different policies available for the users to choose depending on their needs. (3) The Nebula Central will inform the user which data nodes are selected (in a certain order) to upload the files. (4) The files will be directly uploaded to the selected data nodes from their locations. Note that if multiple files from different locations are uploaded, each file may be uploaded to a different data node. (5) Once the files have been successfully uploaded to each of the selected nodes, the data nodes will send update information to the DataStore Master. The DataStore Master will keep the metadata of the files and replicate the files depending on the replication policy.

Data node selection: Nebula supports different data node selection policies for uploading external files into Nebula. The simplest policy is *Random* node selection policy, which returns a random set of data nodes for uploading. Although *Random* data node selection is simple and may provide load balancing, this may result in a poor performance due to network constraints. Another possible data node selection policy would be to select data nodes which are geographically close to the file's location (*Close-distance* policy). This policy may be useful if the user is concerned about geographic proximity of their data in Nebula, however, it may not yield a good overall performance for data processing.

Selecting data nodes with a low latency (*Low-latency* policy) and/or high download bandwidth to the source is likely to improve the data upload performance, and is desirable when the user wants to have a low latency access to the data or to upload their data quickly into Nebula. However, if the data is uploaded for further processing, uploading the data quickly to Nebula may not always result in a good overall processing time since it is possible that the network bandwidth between the selected data nodes and compute nodes that eventually compute on the data is congested. This may lead to overall application performance degradation. In such scenario where locality between data nodes and compute nodes is important, we propose a *Compute-aware* data node selection policy which sorts a set of data nodes based on the network bandwidth proximity between data nodes and compute nodes. This method provides a high bandwidth locality awareness of data nodes with respect to compute nodes for uploading which will improve the overall application performance.

Both the low latency and high bandwidth information for locality-aware data upload can be obtained from the DataStore Master as discussed in 3.2.3. All of these different data upload options are exposed to the end users. Thus, users can choose any of these options depending on their needs. If the main goal is to have a low latency access to the data, they can choose to upload the data using the *Low-latency* policy. On the other hand, if the goal is to upload and process the data for data-intensive analytic applications, the *Compute-aware* policy would be the best option. Once the data is uploaded, the DataStore Master will create multiple copies of the data and distribute them to other data nodes to ensure data availability. We will discuss the data availability issue later in Section 3.2.5.

Block size: Large files in Nebula may be partitioned into blocks of fixed size. The block size is the same as that of the block granularity processed by a compute node. The advantage of partitioning files into the same size blocks is to minimize the latency in a limited network bandwidth environment during data transfer by enabling parallelism over multiple data nodes. Another advantage of splitting a file and storing each block to a different data node is to avoid performance bottleneck within a single data node. While partitioning a file into blocks provides advantages, file partitioning also has some costs. Selecting the appropriate size of a block should take the following costs into consideration. Splitting files into blocks and storing them on multiple data nodes will increase the amount of metadata in the DataStore Master. The smaller the block size, the greater the amount of the metadata. The default block size is set to 64 MB, however this can be tuned by the clients.

3.2.5 Data Availability

Volunteer data nodes in Nebula that may exit the system at any time. In order to ensure the availability of every file stored in DataStore, Nebula supports file replication. By default, Nebula keeps 3 copies of each file and distribute them across different data nodes. Further, as data nodes come and go, the DataStore Master needs to ensure that a certain number of copies remain online at all times. When a data node storing a specific file withdraws from the system, the DataStore Master will create a new copy of the file to maintain the number of replicas available.

In addition to keeping each file available, the DataStore Master may also implement a data node selection policy that determines which nodes should store each of the copies. For example, the DataStore Master can distribute the copies in different part of the globe for read-only data that is frequently accessed worldwide in order to provide low latency to all users. On the other hand, if the data is frequently modified, the DataStore Master may store the copies close to each other for better consistency. The DataStore Master can also distribute the copies to data nodes based on their loads in order to achieve a good load balancing between the data nodes. In the future, we would like to further explore a more dynamic policy to maintain high data availability with lower overhead, specifically for volunteer-based storage systems.

3.3 ComputePool

The ComputePool provides computational resources to Nebula applications. Each application is provided a ComputePool which is a subset of the volunteer compute nodes. The ComputePool is managed by a ComputePool Master which manages the volunteer compute nodes, assigns tasks to them, and monitors their health and execution status.

3.3.1 Compute Node

Compute nodes are volunteer nodes that carry out computation on behalf of an application. All computations in Nebula are carried out within the Native Client (NaCl) sandbox [15] provided by the Google Chrome browser. The use of NaCl gives several advantages. First, one of the biggest concerns of donating computational resources in voluntary computing is the security of the volunteer node. The volunteer node must be protected from malicious code and prevent such code from hurting the volunteer. NaCl is a sandboxing technology that executes native code in a web browser, and allows users to execute untrusted code safely by restricting privileges and isolating faults from the rest of the system. This provides users a secure way to donate computational resources. Second, it makes it easy for users to join Nebula, the users do not need to download any additional software to donate their resources. All it takes to join Nebula for a Chrome user is to enable the NaCl plugin and point the browser to Nebula Central. Third, using NaCl has minimal performance overhead, since it can execute native code. The sandbox adds little overhead to the application, and execution times inside and outside the sandbox are comparable [15]. The NaCl sandbox does have a constrained memory-space, and current applications must be designed to fit inside of it, although future designs can mitigate the risk by moving data to disk.

The compute node is a general-purpose implementation, and executes any application properly compiled into NaCl executables. The compute node executes within the browser by initially downloading a generic Nebula-specific Web page from Nebula Central. This page has Javascript code that carries out some initialization functions and then points the node to the corresponding ComputePool Master URL. Each compute node then contacts the ComputePool Master periodically and asks for a task by downloading a Web page from the ComputePool Master corresponding to a task. This page contains Javascript code that is responsible for downloading and invoking the application-specific NaCl code, and communicating with the ComputePool Master. The Javascript of the page downloads the NaCl code (`*.nmf` and `*.nexe` or `*.pexe` files) for the task from the ComputePool Master (or uses a cached copy if already downloaded for a previous task), and listens to when the NaCl module is fully loaded. It then sends task-specific options to the NaCl code in a message as a JSON object. The NaCl code listens for messages, parses the options, and downloads the input data from the DataStore. It then carries out the application-specific computation and uploads the output files back to the DataStore. Finally, it passes control to the Javascript layer, which reports back the task completion status and any other metadata to the ComputePool Master.

Fault tolerance: The compute node can handle a variety of failures such as failure to load, failure to launch the executable, crash after starting, and so on. The NaCl code sends heartbeat messages to the Javascript layer, which cancels the task if it does not receive heartbeat messages often enough. The compute node then asks for another task from ComputePool Master while noting the current task failed. In most cases, the task will be restarted, either by the same node or another node. In the case of slow data transfers, the ComputePool Master checks its recent history to see if multiple nodes have reported an issue with the same data node. If so, it can re-execute the tasks with their input data on that data node, or use the copy of the file stored in a different node if it's available, or ask the DataStore Master to move the data to a different data node. If the input data does not exist in the DataStore, Nebula will recreate the data if it was produced by an earlier task execution.

3.3.2 ComputePool Master

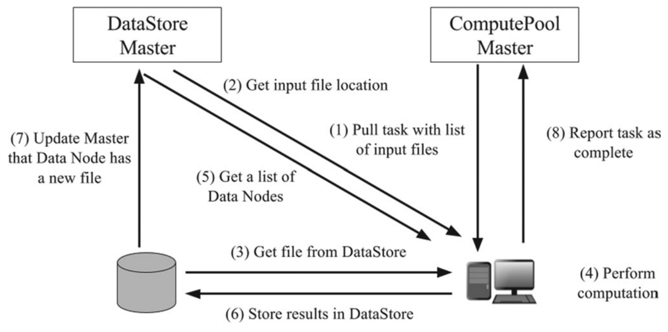
The ComputePool Master is responsible for the health and management of the compute nodes. It provides a web-based interface to allow its constituent compute nodes to download job executables provided via Nebula Central. When the Nebula Central instantiates a ComputePool Master for an application, it passes a generic Javascript code, application-specific NaCl executables as well as application and job input parameters, to it. The ComputePool Master uses this information to generate a set of tasks for the jobs. When it assigns a task, it creates a web page with these arguments built into the actual source of the page-this page is downloaded and executed by a compute node, as discussed above.

The most important function of the Master is the scheduling of tasks to compute nodes. The ComputePool Master binds to a scheduler selected by the application writer. To support the scheduler, the ComputePool Master implements a dashboard which collects per-node execution statistics and the inter-node link bandwidths tracked by the Nebula Monitor. The scheduler uses monitoring information to decide which tasks should be run and where it should be run on. The complexity of scheduling decisions can be as simple as a random scheduler, to more complex as our adaptive locality-aware MapReduce scheduler which attempts to approximate network between nodes and computation abilities of each compute node to complete a task in minimal time despite nodes going up and down during execution. The scheduler runs periodically updating node to task mappings and when nodes check in for work they are allocated the assigned task(s).

3.3.3 Locality-Aware MapReduce Scheduler

As an example, we present a basic locality-aware scheduling algorithm that will be used for MapReduce applications:

1) Get updates from the Resource Manager, DataStore Masters, and the Nebula Monitor about the current state of the system.
2) Estimate the time to completion for each running task. If this is an underestimate, we adjust the projected running time to account for the inaccuracy and therefore rank this node lower for scheduling.
3) For each new task, create estimates for each (node, task) pair. This estimate is based on (1) the download time for all of the task inputs from the DataStore, (2) the time to execute the task, and (3) the time to store the results back to the DataStore. The communication estimates assume locality awareness as the nearest data nodes (in terms of bandwidth, latency, etc.) will be selected for inputs and outputs.
4) For each (node, task) pair, we estimate its finishing time based on the estimated remaining completion time of the currently running task on the node plus the estimate of subsequently running the task on that node. We then order these pairs in a priority queue based on this estimated finishing time. We also randomly insert (task, node) pairs for new nodes that we have not yet seen previously (i.e., have no estimates) to allow the system to use them and to learn their capabilities.
5) For each task, we select the best node from the prior step estimated to complete a task. To prevent a high-speed node from attracting too many concurrent tasks, we cap the number of tasks per node at 2 in each iteration of the scheduler. Note: this does not mean a node is limited to executing 2 tasks during the lifetime of a run.


DataStore Master
ComputePool Master
(2) Get input file location
(1) Pull task with list of input files
(7) Update Master that Data Node has a new file
(5) Get a list of Data Nodes
(8) Report task as complete
(3) Get file from DataStore
(4) Perform computation
(6) Store results in DataStore
Fig. 3. Control and data flow in executing task in Nebula.

3.3.4 Task Execution

Once an application is injected into the system via Nebula Central and the input data has been placed within the DataStore, the compute nodes are ready to accept and execute tasks. Fig. 3 shows the various steps involved in the execution of a task. (1) The compute node contacts the ComputePool Master scheduler periodically and asks for tasks. The scheduler would assign tasks to the compute node based on its scheduling policy. (2-3) The compute nodes would then download the application code from the ComputePool Master, and the input data from the DataStore nodes. (4) The execution starts inside NaCl as soon as the downloads complete. (5-6) Once execution finishes, the outputs are uploaded back to the DataStore and (7) the data nodes would update the DataStore Master. (8) Finally, the ComputePool Master will be notified upon the job completion.

The size and composition of the ComputePool and DataStore are application-specific. In addition, how the compute nodes are allocated across ComputePool is controlled by the Resouce Manager which will be discussed later.

3.3.5 Load Balancing and Locality Awareness

To achieve load balancing, the ComputePool Master scheduler takes into account the CPU speeds and current loads on different compute nodes to assign them tasks. To achieve locality awareness, it can assign tasks to compute nodes based on the location of the input data for the tasks. The scheduler maintains a list of preferred tasks for each compute node based on its locality, and assigns tasks from this list when the compute node requests additional work. This locality-based task allocation mechanism, along with the locality awareness support in the DataStore, reduce network overhead substantially.

3.3.6 Fault Tolerance

The ComputePool Master allows re-execution of tasks to achieve fault tolerance in the face of compute node failures. Fault tolerance to soft failures can be handled by a compute node itself, as discussed above. If a compute node becomes unresponsive during the execution of a task by timing out for a certain duration, the ComputePool Master reassigns the unfinished task to another compute node. The timeout value is set large enough to allow for transient failures or missed heartbeats, so that resource wastage can be avoided if a node becomes responsive again quickly and indicates that it is making progress on the task.

3.4 Nebula Monitor

The Nebula Monitor does performance aggregation of compute and data nodes. It accepts queries from the other Nebula services such as DataStore Master and ComputePool Master for pair-wise bandwidth and latency statistics. These statistics are used by the other services to make decisions including scheduling compute tasks, combining inputs for tasks, and almost all DataStore decisions.

The Nebula Monitor uses a simple moving average of its metrics. In the future we would like to combine information, such as multiple simultaneous requests, to better predict the operation of a node and to better note when performance is lagging to take immediate action (e.g., moving data from a node, changing destination nodes of a running task).

The Nebula Monitor also measures certain metrics like bandwidth, latency, etc. between the compute nodes and Nebula Central that is used as the default value if the link-to-link data is not known. It first tries to determine the physical location of the compute nodes by using the GeoLocation API provided by the browser. If this fails, it falls back to a IP to location lookup on the server-side.

3.5 Resource Manager

The Resource Manager allows multiple applications from different ComputePools to run concurrently which is similar to those in the traditional centralized cloud [21], [22], [23], [24]. It manages all the compute nodes in the system. Having a global view of the compute nodes, the Resource Manager is responsible for sharing the compute nodes among ComputePools by dynamically gives a subset of available compute nodes to each ComputePool.

The Resource Manager in Nebula uses an optimistic concurrency control with a lease-based mechanism [25]. In this mechanism, each ComputePool Master must lease a set of nodes from the Resource Manager before using them. A lease in this mechanism has an expiration time associated with it which provides a guarantee that the nodes will be held by a ComputePool no longer than the lease time (with a possibility of some grace period). Thus, every ComputePool's scheduler needs to estimate the lease time before requesting nodes. This lease estimation can be computed based on the historical statistics for recurring jobs or using a combination of data transfer time (data size over the bandwidth between nodes) and the average computation time of processing similar tasks. After the lease time expiry, the Resource Manager will reclaim the resources, make them available to the other ComputePools and potentially clean up any tasks on the expired lease. The lease information for every node is shared to every ComputePool's scheduler to allow them trade-off between waiting for the desired, but currently busy, nodes and the less-desired available nodes. Consider a case where a scheduler needs to schedule a job whose inputs are originated in the US but the only available compute nodes are located in Asia. In this case, the scheduler may decide to delay scheduling the job and wait for busy compute nodes that are located in the US if it knows that they will be available in the near future.

The Resource Manager may also implement any policies that must be applied by any ComputePools. For example, the Resource Manager may limit the number of nodes that can be acquired by a ComputePool at any given time for proportional sharing. Another policy could reclaim any resources from a ComputePool that acquired too many nodes to avoid resource starvation for other ComputePools.

4 MapReduce on Nebula

We have implemented a MapReduce framework on Nebula. MapReduce [12] is a popular programming paradigm for large-scale data processing on a cluster of computers, and provides a simple programming model for various data-intensive computing applications. For this reason, we have selected MapReduce as our first Nebula framework to test of our approach. Note that our Nebula MapReduce framework is not based on existing MapReduce implementations as in Hadoop MapReduce; rather it is implemented on top of the Nebula infrastructure and utilizes DataStore and ComputePool for all storage and compute needs respectively.

A Nebula MapReduce application consists of two jobs: map and reduce. The reduce job is dependent on the map job, so it cannot start execution until the map job completes. The map and reduce jobs are written in C++ and compiled against the NaCl compilers. The output of the compilation phase is a set of `.nexe` or `.pexe` files which can be uploaded to Nebula Central for distribution via its web interface. This process only needs to be done once for the subsequent application instances that use the same executable. The input data of a MapReduce application needs to be pushed to the DataStore first and all of these inputs will share the same namespace. A user will then need to post a MapReduce application and instantiate it by providing parameters including the number of map and reduce tasks, namespace, and a list of inputs.

A MapReduce scheduler in Nebula was designed to optimize MapReduce task execution and data movement and is used by the ComputePool and DataStore for computation and data respectively. The former uses a greedy heuristic that chooses the fastest nodes to complete each task exploiting available nodes, even if this creates the occasional duplicate task execution. This method has been shown to run the vast majority of tasks quickly while preventing the often-occurring long-tail of MapReduce execution.

Once a MapReduce application has been instantiated, the MapReduce scheduler can start assigning tasks to compute nodes. To execute a map task, the compute node would download the input data from the DataStore and the executable file from the ComputePool Master. The compute node would then execute the map task, write its outputs to the DataStore, and report back to the ComputePool Master, so that it can log the progress of the MapReduce job. The execution of a reduce task is very similar to that of the map task, except that a reduce task takes in different inputs, mainly a list of dependencies instead of input files and the reducer number. These inputs ensure that a reduce task cannot start executing until all its dependencies (map tasks) have finished executing. Once the dependencies are finished, each reduce task can start executing by reading its input data, running the reduce function, and outputting its results to the DataStore. Completion of the reduce task is then reported back to the ComputePool Master.

4.1 Example: Word Count Application

We implemented a MapReduce Wordcount application which counts the total frequency of each word appearing in documents. Listing 1 shows the main map and reduce code in Nebula. Each map task obtains its input data from the DataStore, performs Wordcount algorithm on it which results in a list of key-value pair where the key is the word and the value is the frequency of the word. The list is then partitioned into as many output files as there are reduce tasks by using a SHA1 hash. All map outputs are uploaded back to the DataStore at the end of each map task.

A reduce task downloads the map outputs corresponding to its partition from the DataStore and aggregates the count of each word. The result is again a list of key-value pairs where the value is the total number of times the word appeared in the documents. The final output is then uploaded back to the DataStore.

Listing 1. Nebula MapReduce WordCount code

```
class Map: public Nebula::MapNebulaTask{
  void map (string data){
    string word;
    wordstream wss(data);
    while (wws > > word){
      emit(word, "1");
    }
  }
  void handleAllDownloadsFinished (){
    // handle intermediate data
  }
}
class Reduce: public Nebula::ReduceNebulaTask{
  map<string, vector<string>> wordcounts;
  void handleDownloadData (string data){
    // download the intermediate data
  }
  void reduce (string word, vector<string> counts){
    int count = 0;
    for(i=counts.begin(); i != counts.end(); i++){
      count += atoi(i);
      emit (word, NumberToString(count));
    }
  }
  void handleAllDownloadsFinished (){
      // handle the output of the reducer
  }
}
```

5 Evaluation

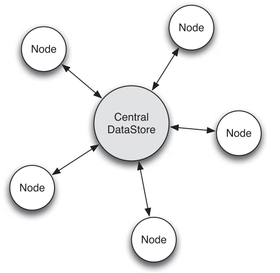
5.1 Experimental Setup

We set up 52 nodes on PlanetLab [13] Europe (PLE), each with Google Chrome and other required software packages installed. We limit ourselves to PLE instead of the entirety of PlanetLab due to software requirements of Google Chrome. These nodes are located in 15 different countries and have bandwidth ranging from 256 Kbps to 32 Mbps. All the dedicated Nebula services (Nebula Central, DataStore Master and ComputePool Masters, and Nebula Monitor) are hosted on a single machine with Dual Core Pentium E2200 @ 2.4 GHz, 4 GB RAM, 150 GB Hard Drive, running Ubuntu 12.04 Linux. It also hosts MySQL databases to support Nebula Central and ComputePool Masters (2.2 GB), and Redis databases (800 MB) to support Nebula Central and DataStore Masters. In most of our experiments, we run a data node and compute node on each available PlanetLab node (this number varies from 38-52 in our experiments due to PlanetLab node failures), except where specified.

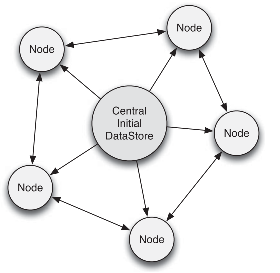
5.2 Comparison with Different Platforms

The first set of experiments directly compare different approaches in a volunteer environment. We perform experiments using the Nebula MapReduce Wordcount applications. Our input dataset consists of a set of 1500 ebooks from Project Gutenberg which amounts to 500 MB of data. We expand and contract this dataset to yield different input dataset sizes for different experiments. We note that *the maximum data size in our experiments is limited due to PlanetLab bandwidth caps*, an issue that we believe should not be a problem in a true volunteer or commercial edge cloud system. Memory limitations also limit the maximum amount of computation available today. We also configure the number of mappers and reducers, with each mapper performing computation over a given number of books. We vary the number of mappers (and thus the number of books per mapper) and reducers during experiments.

For comparison, we emulate two alternate volunteer computing models on top of the Nebula infrastructure as shown in Figs. 4a and 4b: *Central Source Central Intermediate Data (CSCI)* and *Central Source Distributed Intermediate Data (CSDI)*. These models correspond to the data models supported by BOINC [10] and a MapReduce-tuned BOINC version [14] respectively. These models have three key aspects which differentiate them from Nebula. First, in CSCI and CSDI the input data is centralized. The central server is usually the single source of input data. To model this, we use a dedicated host as the single source of data. Second, the decision where to store intermediate data (map's output) is different. In CSCI, intermediate data is stored centrally

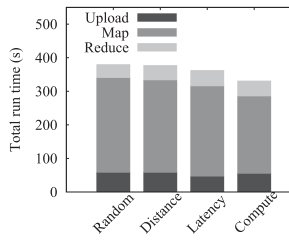


(a) Centralized

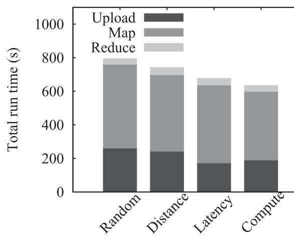


(b) CSDI

Fig. 4. Data flow in centralized and CSDI.



(a) 16MB chunk upload



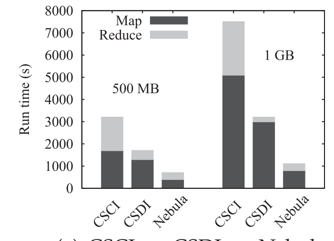
(b) 64MB chunk upload

Fig. 6. MapReduce on different chunk size of external data.

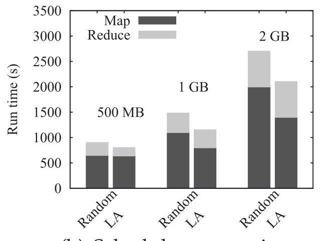
whereas in CSDI it is stored locally on the node that performed the map task. To emulate the CSDI system, we use a configuration where all nodes have both their compute and storage capabilities switched on, and further, the DataStore Master always tries to use a data node already located at the compute node (if one exists). Third, the schedulers are different. In CSCI and CSDI, tasks are assigned randomly without concern for data locality, while the default Nebula scheduler is locality-aware.

The MapReduce Wordcount application was run multiple times on each of the different environments with different size datasets (500 MB, 1 GB, and 2 GB). All experiments used 300 map tasks each, along with 80, 160, and 320 reduce tasks for the 500 MB, 1 GB, and 2 GB input sizes respectively. In this experiment, the data has already been stored randomly across the data nodes. We do not consider the data upload time in this experiment since the size of each ebook, which is the input data in this experiment, is relatively small, thus the upload time is not significant. The CSCI and CSDI systems had a centralized data node with an average upload bandwidth of 4Mbps, which is similar to the bandwidth of many residential and commercial sites.

Fig. 5a compares the Nebula model with the locality-aware scheduler against the CSCI and the CSDI models. The results indicate that the Nebula approach is far superior to these baselines due to the removal of data bottlenecks. For map tasks, Nebula is able to find compute nodes close to the data sources, and outperforms both CSCI and CSDI which rely on a centralized data node. For reduce tasks, both CSDI and Nebula exhibit good performance compared to CSCI, as they retain intermediate data locally. As the data size increases to 1 GB, Nebula continues to perform better (by 580 and 200 percent versus CSCI and CSDI respectively), and the larger data size shows the additional benefit of locality-awareness by selecting compute nodes based on performance estimation.



(a) CSCI vs. CSDI vs. Nebula



(b) Scheduler comparison

Fig. 5. MapReduce across different environments.

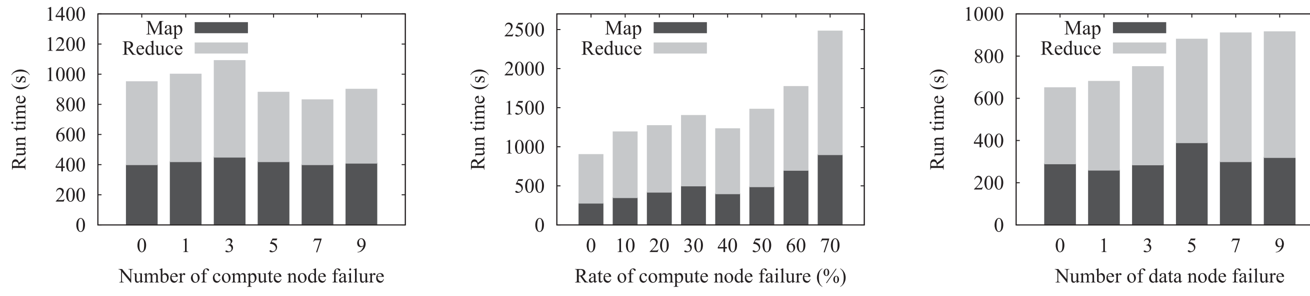
To see the impact of choice of scheduler, we compare the locality-aware (LA) scheduler to a Random scheduler within Nebula, which randomly assigns tasks to compute nodes. Fig. 5b shows the result. We find that the LA scheduler outperforms the Random scheduler by 16-34 percent across the different data sizes. Also, the performance difference between the two schedulers is higher for larger data sizes due to increasing data transfer costs. The benefit of locality is particularly pronounced for the map tasks, which are scheduled close to the input data sources. The reduce tasks benefit less from locality, since they need to download intermediate data from multiple data nodes.

Looking closely to Figs. 5a and 5b, we can see that the Random scheduler in Nebula even outperforms the CSCI and CSDI models by 5X and 2X respectively. The main reason of this improvement is because the two models suffer from bandwidth contention since the output bandwidth of the data node that stored the data is multiplexed to service data requests from the compute nodes.

5.3 External Data Upload

In this experiment, we consider a bigger size for each file (16 MB and 6 4MB per input file). We compare the performance of 4 data node selection policies discussed in Section 3.2.4. The performance includes the file upload time and job running time. We uploaded a 512 MB file from the US to data nodes located in Europe. The first data node selection policy is *Random*, which selects data nodes for each file chunk. Note that multiple chunk files could be uploaded to a single data node. The second data nodes selection policy is *Distance*, which upload the data based on the closeness in geographic distance between a data node with the file's location. The third policy is *Latency* which uploads the file to a data node that minimize the total upload time. The last method is *Compute* which is the compute-aware data upload policy that combines the network performance (1) between the source and the data nodes, and (2) between the data nodes and the compute nodes.

Figs. 6a and b show the average total running time of MapReduce WordCount applications on 16 MB and 64 MB input chunk size, respectively. As we can see, the compute-aware policy results in the best overall performance even if its upload time is slightly higher than the *Latency* policy which mainly focused on uploading the data from the source to the data nodes. The reason is that the compute-aware upload policy considers the proximity bandwidth used in transferring data from the storage nodes to the compute nodes which is used by the scheduler in selecting compute nodes to process the data.



(a) Compute node failures (b) Compute node failure rate (c) Data node failures

Fig. 7. Performance in the presence of node failures.

5.4 Fault Tolerance

In the next set of experiments, we induce crash failures in both the compute and data nodes during execution. Nebula has mechanisms to provide fault tolerance in the presence of node failures. These mechanisms include re-execution of tasks to handle compute node failures and data replication to handle data node failures. The goal of these experiments is to show the robustness of the Nebula infrastructure in the presence of such failures, even if it comes with a performance cost. We note that there were a large number of background transient failures even in the previous set of performance experiments that we are already robust to. For instance, for the 500 MB experiment with Nebula-LA (Figs. 5a and 5), we had a total of 1557 transient failures during the run (364 process restarts, 1146 NaCl execution failures, and 47 data transmission failures), which the system recovered from.

All of these experiments use the Nebula system setup with the Nebula-LA (locality aware) scheduler. We use a 500 MB input data size, along with 300 map and 80 reduce tasks for all experiments. We note that we *actually kill* the associated processes for these tests, rather than simulating the failures, so these experiments illustrate the robustness of Nebula in the presence of real-life failures.

5.4.1 Compute Node Failures

We first show the impact of compute node failures. In these experiments, we do not fail the data nodes, and do not use data replication for any of the data in the system. In the first experiment, we kill a subset of compute nodes part-way (50 percent) through the execution of the map phase, and these nodes are considered failed for the remaining duration of the experiment. As a result, the progress of the tasks running on the failed nodes is lost, and these tasks are re-executed by the ComputePool Master on other nodes, once the failures are detected. Fig. 7a shows the results of inducing these failures. We find that the system is relatively stable in the presence of a moderate number of node failures (compared to a more severe scenario considered next). This is because the Wordcount application in these experiments is input data bandwidth limited, and there are sufficient computational resources available to carry out the computation even with the induced failures.

In the next experiment, we induce more severe failures, failing a large part of the system throughout the run. In this case, we randomly kill a subset of compute nodes, but allow failed nodes to come back up after a period of about 90 seconds to ensure that a fixed proportion of nodes in the system are failed at all times. Fig. 7b shows the results of inducing these failures. In this case, we see that beyond a certain rate of failures (over 50 percent), the runtime deteriorates. At this point there are so many compute failures that the system starts becoming compute-limited, and has to re-execute many unfinished tasks.

5.4.2 Data Node Failures

Next, we show the impact of data node failures. In the first set of experiments, we do not use data replication for intermediate data. The input data is still replicated twice, and data node failures are induced such that a copy of the map input data will always be available. However, a data node failure will result in the loss of intermediate data needed for reduce tasks, leading to the re-execution of map tasks needed to recreate this data (this re-execution time is attributed to reduce time in the results).

In this experiment, we kill a subset of data nodes partway (50 percent) through the execution of the map phase. As a result, intermediate data generated by already finished map tasks uploaded to the failed data nodes is lost, and has to be recreated by re-executing the corresponding map tasks. Fig. 7c shows the results of inducing these failures.

The results show that the total runtime increases as we increase the number of data node failures, because of more map task re-executions, though the system is robust to such failures and is able to complete the runs. However, for this reason, runtime increases with failure rate unlike the compute node failure scenarios in Fig. 7a.

5.5 Scalability

Being a voluntary computing system, a future Nebula deployment could consist of thousands of volunteer nodes that are connected to the system simultaneously. Considering this, one of the design goals is that centralized components can scale up as required. More importantly, as the amount of data, the number of nodes, and the geographic spread increases, the system should be able to take advantage of the added number of resources, the greater amount of parallelism, and more choices for locality-aware resource allocation, without bottlenecks forming.

Fig. 8a shows the performance of Nebula as we increase the number of compute and data nodes in the system from

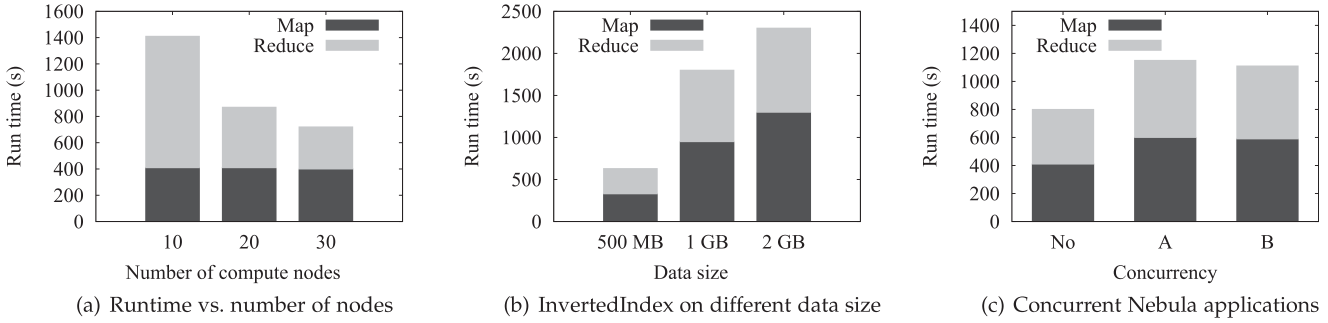


Fig. 8. Scalability and concurrent applications in Nebula.

10-30 each (i.e, 30 refers to 30 compute nodes and 30 data nodes). All these experiments use an input data size of 250 MB, along with 250 map and 80 reduce tasks. The input data is kept on 8 data nodes, however, all compute nodes can participate in the computation and all data nodes can store intermediate data.

We see that the runtime decreases as the number of nodes increases. In particular, we see that while the map time remains the same due to the bandwidth constraints of the input data nodes, the reduce time decreases with the increasing size of the system. This is due to greater compute and data parallelism, as well as the availability of more nodes to select compute nodes with high locality to the intermediate data.

To see the scalability of Nebula in the presence of large intermediate data as well, we present results for another MapReduce application, InvertedIndex. The InvertedIndex program creates an index to identify which files contain which words. The output is often used as one of the steps to enable fast searching through text documents. As opposed to WordCount which provides high reduction of input data, InvertedIndex is characterized by an expansion of the input data. This set of experiments illustrates the scalability of Nebula in the presence of large intermediate data as well. Fig. 8b shows the performance of Nebula as it creates an inverted index from 500 MB, 1 GB, and 2 GB text files. We see that the system still scales with InvertedIndex as with Wordcount as the problem size expands.

Lastly, we consider running multiple concurrent applications on Nebula. Fig. 8c shows the performance of two concurrent Wordcount applications running in Nebula, as compared to a single application. These experiments use an input data size of 250 MB, along with 250 map and 80 reduce tasks. As expected, the performance of each application (Concurrent-A and Concurrent-B) is worse than that of a single application running in the system. However, their performance is similar to each other. In this result, Nebula simply multiplexes its resources equally among concurrent applications, however, any resource sharing policy may be added in the Resource Manager as shown in [25].

6 Related Work

Volunteer edge computing and data sharing systems are best exemplified by Grid and peer-to-peer systems including, Kazaa [26], BitTorrent [11], Globus [27], Condor [28], BOINC [10], and @home projects [29]. These systems provide the ability to tap into idle donated resources such as CPU capacity or aggregate network bandwidth, but they are not designed to exploit the characteristics of *specific* nodes on behalf of applications or services. Furthermore, they are not designed for data-intensive computing.

Other projects have considered the use of the edge, but their focus is different. [19], [30], [31] focus on minimizing latency in answering queries in geo-distributed data centers and [18], [32] focus on stream analytic in geo-distributed system. Cloud4Home [33] focuses on edge storage where Nebula enables both storage and computation critical to achieving locality for data-intensive computing. Other storage-only solutions include CDNs such as Amazon's CloudFront that focus more on delivering data to end-users than on computation.

There are a number of relevant distributed MapReduce projects in the literature [34], [35], [36], [37]. Moon [34] focuses on voluntary resources but not in a wide-area setting. Hierarchical MapReduce [35] is concerned with compute-intensive MapReduce applications and how to apply multiple distributed clusters to them, but uses clusters and not edge resources. [36] focuses more on cross-phase Map Reduce optimization, albeit in a wide-area setting.

There are also projects that look at the issue of proving reliable data storage in non-datacenter environments including P2P systems, volunteer-based systems, and utility-like data storage systems [38], [39], [40]. Although this work is relevant to the fault tolerant aspect of the storage system in Nebula, we use a simple static file replication to ensure that files remain available in the DataStore. One possible solution that is more efficient is discussed in the next section.

7 Open Research Challenges

We believe that the wide-area and the voluntary aspect of an edge cloud presents some opportunities and research challenges in deploying a reliable system such as Nebula.

7.1 Compute Node Reliability

Stragglers are likely to occur unpredictably in such a system due to the variety of computational power of volunteer nodes and the dynamic WAN performance. One possible solution is to adapt the reliability-based technique [41] by combining the notions of timeliness and correctness which results in a decrease in overall time-to-completion while increasing confidence in the results. This technique can be adapted in the Nebula's Compute-Pool Master scheduler by:

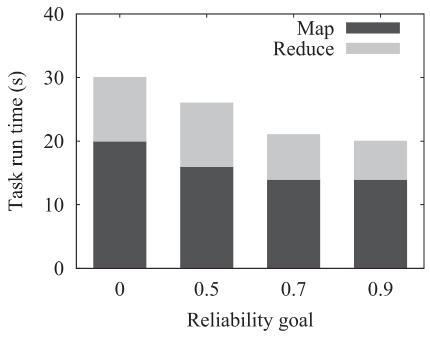


Fig. 9. Average task completion time.

- Establishing the notion of *reliability score* that is maintained per node.
- Scheduling a task to a group of compute node instead of only one node.
- Selecting nodes to meet the *reliability goal* based on the combined reliability scores of the individual node.

A node's reliability score can be computed from the percentage of completed tasks within the approximate deadlines. Let r_n represent the reliability score of a compute node n, b_{max} represents the maximum deviation that we are willing to tolerate, $\hat{r}$ be the sample mean, and s represents the corrected sample standard deviation. If $r_n < \hat{r} - b_{max}s$, then the compute node n will be ignored during scheduling due to its lack of reliability. If $r_n > \hat{r} + b_{max}s$, then we can set $s_n = \hat{r} + b_{max}s$ to avoid using reliability scores that are unrealistically high. Very high scores are likely an artifact of our scoring mechanism and not a reflection of reality.

We performed initial experiments by running the standard Nebula MapReduce Wordcount application with a 128 MB input file split into 10 MB chunks that were randomly distributed across the data nodes. We enforced a minimum of 2 compute nodes per reliability group and a maximum of 6. We compared the original scheduler that does not consider the reliability of the nodes to the modified one using reliability goals of 0.5, 0.7, and 0.9.

Fig. 9 indicates that the use of a higher reliability goal results in a decrease in the average task runtime because it is less susceptible to stragglers. However, this may come at the cost of increasing the number of resources used per task, which can limit the number of distinct tasks that can be executed concurrently.

7.2 Data Availability

Using a fixed number of replicas for every file may not be an appropriate solution given the dynamic nature and heterogeneity of a volunteer-based storage system. We propose a possible solution by measuring the availability of individual node and ensuring that every file's availability score meets the minimum threshold. A file's availability score itself can simply be a combination of every node's availability score where the file and the replicas are stored. A node is history of being online is used to compute its observed probability of being offline $\widehat{P}_{down,i}$. In order to choose a set S of nodes to replicate the file onto, new nodes are added until the following inequality holds: $1 - \prod_{i \in S} \widehat{P}_{down,i} \geq P_{avail}$ Alternatively, we may use a global availability score $\widehat{P}_{down}$ for all nodes instead of maintaining per-node scores. This is a simplification, but in environments where adequate history cannot be found for most nodes (and default values are used as placeholders), such a policy can be superior to one using unreliable per-node scores. With this global availability score, we can simply compute the number of nodes n necessary to meet the availability goal since we can solve that $n = \lceil \frac{log(1-P_{avail})}{log(\widehat{P_{down}})} \rceil$ Choosing nodes then becomes a matter of selecting which n nodes will be used to store the data and its replicas which is an orthogonal problem.

8 Conclusion and Future Work

We presented the design and implementation of Nebula, an edge-based platform that enables distributed in-situ data-intensive computing. The Nebula components were described along with abstractions for data storage, DataStore, and computation, ComputePool. A working Nebula prototype runs across edge volunteers on the PlanetLab testbed. An evaluation of MapReduce on Nebula was performed and compared against other edge-based volunteer systems. The locality-aware scheduling and data placement enable Nebula MapReduce to significantly outperform existing systems. In addition, we showed that Nebula is highly robust to both transient and crash failures.

Future work includes topics discussed in Section 7 such as improving the compute nodes reliability in a volunteer-based system and explore a more efficient policy to the static file replication that would maintain the high availability of files stored in the storage system. Furthermore, we plan to expand the range of applications and frameworks that can be ported to Nebula and validate our initial findings on a much larger scale.

Acknowledgments

The authors would like to acknowledge Muhammed Uluyol for his contributions on the compute node reliability and data availability techniques. We also acknowledge grant NSF CSR-1162405 and CNS-1619254 that supported this research.

References

[1] Amazon web services. [Online]. Available: http://aws.amazon.com
[2] Microsoft azure. [Online]. Available: http://azure.microsoft.com/
[3] Amazon data center location. [Online]. Available: http://docs.aws.amazon.com/general/latest/gr/rande.html
[4] Microsoft data center location. [Online]. Available: https://www.microsoft.com/en-us/cloud-platform/global-datacenters
[5] Google data center location. [Online]. Available: https://www.google.com/about/datacenters/inside/locations/
[6] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. V. Madhyastha, "Spanstore: Cost-effective geo-replicated storage spanning multiple cloud services," in *Proc. 24th ACM Symp. Operating Syst. Principles*, 2013, pp. 292–308.
[7] J. C. Corbett, et al., "Spanner: Googles globally distributed database," *ACM Trans. Comput. Syst.*, vol. 31, no. 3, 2013, Art, no. 8.
[8] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for VM-based cloudlets in mobile computing," *IEEE Pervasive Comput.*, vol. 8, no. 4, pp. 14–23, Oct.–Dec. 2009.
[9] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Proc. 1st Edition MCC Workshop Mobile Cloud Comput.*, 2012, pp. 13–16.
[10] D. P. Anderson, "Boinc: A system for public-resource computing and storage," in *Proc. 5th IEEE/ACM Int. Workshop Grid Comput.*, 2004, pp. 4–10.

[11] B. Cohen, "Incentives build robustness in bittorrent," in *Proc. Workshop Economics Peer-to-Peer Syst.*, 2003, vol. 6, pp. 68–72.
[12] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.
[13] B. Chun, et al.,"Planetlab: An overlay testbed for broad-coverage services," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 33, no. 3, pp. 3–12, Jul. 2003.
[14] F. Costa, L. Silva, and M. Dahlin, "Volunteer cloud computing: MapReduce over the internet," in *Proc. IEEE Int. Symp. Parallel Distrib. Process. Workshops Phd Forum*, 2011, pp. 1855–1862.
[15] B. Yee, et al., "Native client: A sandbox for portable, untrusted x86 native code," in *Proc. 30th IEEE Symp. Security Privacy*, 2009, pp. 79–93.
[16] F. Dabek, R. Cox, F. Kaashoek, and R. Morris, "Vivaldi: A decentralized network coordinate system,' *ACM SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 4, 2004, pp. 15–26.
[17] R. Cox, F. Dabek, F. Kaashoek, J. Li, and R. Morris, "Practical, distributed network coordinates," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 1, pp. 113–118, 2004.
[18] 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 *Proc. 11th USENIX Symp. Netw. Syst. Des. Implementation*, 2014, pp. 275–288.
[19] A. Vulimiri, C. Curino, B. Godfrey, K. Karanasos, and G. Varghese, "WANalytics: Analytics for a geo-distributed data-intensive world," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, 2015, 1087–1092.
[20] R. Viswanathan, G. Ananthanarayanan, and A. Akella, "Clarinet: Wan-aware optimization for analytics queries," in *Proc. 12th USENIX Symp. Operating Syst. Des. Implementation*, 2016, pp. 435–450.
[21] B. Hindman, et al., "Mesos: A platform for fine-grained resource sharing in the data center," in *Proc. 8th USENIX Conf. Netw. Syst. Des. Implementation*, 2011, vol. 11, pp. 22–22.
[22] V. K. Vavilapalli, et al., "Apache hadoop yarn: Yet another resource negotiator," in *Proc. 4th Annu. Symp. Cloud Comput.*, 2013, Art. no. 5.
[23] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: Flexible, scalable schedulers for large compute clusters," in *Proc. 8th ACM Eur. Conf. Comput. Syst.*, 2013, pp. 351–364.
[24] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *Proc. 10th Eur. Conf. Comput. Syst.*, 2015, Art. no. 18.
[25] A. Jonathan, A. Chandra, and J. Weissman, "Awan: Locality-aware resource manager for geo-distributed data-intensive applications," in *Proc. IEEE Int. Conf. Cloud Eng.*, 2016, pp. 32–41.
[26] S. Saroiu, K. P. Gummadi, R. J. Dunn, S. D. Gribble, and H. M. Levy, "An analysis of internet content delivery systems," *ACM SIGOPS Operating Syst. Rev.*, vol. 36, no. SI, pp. 315–327, 2002.
[27] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke, "The physiology of the grid," *Grid Comput.: Making Global Infrastructure Reality*, pp. 217–249, 2003.
[28] M. J. Litzkow, M. Livny, and M. W. Mutka, "Condor-a hunter of idle workstations," in *Proc. 8th Int. Conf. Distrib. Comput. Syst.*, 1988, pp. 104–111.
[29] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer, "Seti@ home: An experiment in public-resource computing," *Commun. ACM*, vol. 45, no. 11, pp. 56–61, 2002.
[30] Q. Pu, et al., "Low latency geo-distributed data analytics," in *Proc. ACM Conf. Special Interest Group Data Commun.*, 2015, pp. 421–434.
[31] K. Kloudas, M. Mamede, N. Preguiça, and R. Rodrigues, "Pixida: Optimizing data parallel jobs in wide-area data analytics," in *Proc. VLDB Endowment*, 2015, vol. 9, no. 2, pp. 72–83.
[32] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer, "Network-aware operator placement for stream-processing systems," in *Proc. 22nd Int. Conf. Data Eng.*, 2006, pp. 49–49.
[33] S. Kannan, A. Gavrilovska, and K. Schwan, "Cloud4home–enhancing data services with@ home clouds," in *Proc. 31st Int. Conf. Distrib. Comput. Syst.*, 2011, pp. 539–548.
[34] H. Lin, X. Ma, J. Archuleta, W.-C. Feng, M. Gardner, and Z. Zhang, "Moon: MapReduce on opportunistic environments," in *Proc. 19th ACM Int. Symp. High Performance Distrib. Comput.*, 2010, pp. 95–106.
[35] Y. Luo and B. Plale, "Hierarchical MapReduce programming model and scheduling algorithms," in *Proc. 12th IEEE/ACM Int. Symp. Cluster Cloud Grid Comput.*, 2012, pp. 769–774.
[36] B. Heintz, A. Chandra, and J. Weissman, "Cross-phase optimization in MapReduce," in *Proc. Cloud Comput. Data-Intensive Appl.*, 2014, pp. 277–302.
[37] C. Jayalath, J. Stephen, and P. Eugster, "From the cloud to the atmosphere: Running MapReduce across data centers," *IEEE Trans. Comput.*, vol. 63, no. 1, pp. 74–87, Jan. 2014.
[38] J. Kubiatowicz, et al., "Oceanstore: An architecture for global-scale persistent storage," *ACM Sigplan Notices*, vol. 35, no. 11, pp. 190–201, 2000.
[39] A. Haeberlen, A. Mislove, and P. Druschel, "Glacier: Highly durable, decentralized storage despite massive correlated failures," in *Proc. 2nd Conf. Symp. Netw. Syst. Des. Implementation*, 2005, pp. 143–158.
[40] R. Bhagwan, K. Tati, Y. Cheng, S. Savage, and G. M. Voelker, "Total recall: System support for automated availability management," in *Proc. 1st Conf. Symp. Netw. Syst. Des. Implementation*, 2004, vol. 4, pp. 25–25.
[41] K. Budati, J. Sonnek, A. Chandra, and J. Weissman, "Ridge: Combining reliability and performance in open grid platforms," in *Proc. 16th Int. Symp. High Performance Distrib. Comput.*, 2007, pp. 55–64.

Albert Jonathan received the BS and MS degrees in computer science from the University of Minnesota. He is working toward the PhD degree in the Department of Computer Science and Engineering, University of Minnesota. His research interests lie in distributed systems. He is a student member of the IEEE.

Mathew Ryden received the MS degree in computer science from the University of Minnesota. He is a software engineer at Google. He is a student member of the IEEE.

Kwangsung Oh is working toward the PhD degree in the Department of Computer Science and Engineering, University of Minnesota. His research interests include distributed system and storage systems. He is a student member of the IEEE.

Abhishek Chandra received the BTech degree in computer science and engineering from the Indian Institute of Technology Kanpur, and the MS and PhD degrees in computer science from the University of Massachusetts Amherst. He is an associate professor in the Department of Computer Science and Engineering, University of Minnesota. His research interests include the areas of operating systems and distributed systems. He is a member of the IEEE.

Jon Weissman reveived the PhD degree in computer science from the University of Virginia. He is a professor in the Department of Computer Science and Engineering, University of Minnesota. His current research interests include distributed systems, high-performance computing, and resource management. He is a senior member of the IEEE.