2014 IEEE International Conference on Cloud Engineering

Nebula: Distributed Edge Cloud for Data Intensive Computing

Mathew Ryden, Kwangsung Oh, Abhishek Chandra, and Jon Weissman
Computer Science and Engineering
University of Minnesota
Minneapolis, MN 55455
{mathew, ohkwang, chandra, jon}@cs.umn.edu

***Abstract*—Centralized cloud infrastructures have become the de-facto platform 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 dispersed-data-intensive applications, where the data may be spread at multiple geographical locations. In this paper, we present Nebula: a dispersed cloud infrastructure that uses voluntary edge resources for both computation and data storage. We describe the lightweight Nebula architecture that enables distributed data-intensive computing through a number of optimizations including location-aware data and computation placement, replication, and recovery. We evaluate Nebula's 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 a BOINC like model.**

***Keywords*—Cloud programming models and tools, MapReduce, Data Intensive, Geo-distributed, Edge, Voluntary**

I. Introduction

Today, centralized data-centers or clouds have become the de-facto platform for data-intensive computing in the commercial, and increasingly, scientific domains. The appeal is clear: clouds such as Amazon AWS and Microsoft Azure offer large amounts of monetized co-located computation and storage well suited to typical processing tasks such as batch analytics. However, many Big Data applications rely on data that is geographically distributed, and is not collocated with the centralized computational resources provided by clouds. Examples of such applications include analysis of user data such as blogs, video feeds taken from geographically separated cameras, monitoring and log analysis of server and content distribution network (CDN) logs, and scientific data collected from distributed instruments and sensors. Such applications lead to a number of challenges for efficient data analytics in today's cloud platforms. First, in many applications, data is both large and widely distributed and data upload may constitute a non-trivial portion of the execution time. 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. Third, the cost to transport, store, and process data may be outside of the budget of the small-scale application designer or end-user.

We propose the use of an *edge cloud* for both computation and data storage to address the first two aspects. The use of edge resources is attractive for two reasons. First, many applications relying on distributed data have characteristics making the edge attractive: large amount of processing (e.g., filtering and aggregation) can be done independently in-situ, which yields significant data compression reducing the data movement costs for any subsequent centralized processing. Secondly, the edge is highly attractive today with the provision of powerful multi-core, multi-node desktop and home machines coupled with increasing amount of high bandwidth Internet connectivity. If cost is an issue, then we advocate the use of volunteer edge resources, otherwise, one could envision using monetized edge resources across CDNs or even ISPs, provided these were equipped to offer computational services. In this paper, we present *Nebula*: a dispersed cloud infrastructure that explores the use of volunteer resources to *democratize* data-intensive computing. In contrast to existing volunteer platforms such as BOINC [1], which is designed for compute-intensive applications, and file-sharing systems such as BitTorrent [4], our Nebula system is designed to support distributed data-intensive applications through a close interaction between both compute and data resources. Moreover, unlike many of these systems, our goal is not to implement a specific resource management policy, but to provide flexibility for users and applications to specify and implement their own policies.

In this paper, we present the lightweight Nebula architecture that enables distributed in-situ data-intensive computing and evaluate its performance on an emulated volunteer platform that spans over 50 PlanetLab [3] nodes across Europe. An early version of Nebula was described in our prior work [2], [19] but did not focus on data-intensive computing and fault tolerance, the subject of this paper. 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. We focus on the systems and implementation aspects of Nebula in this paper. We show how a common data-intensive computing framework, MapReduce, can be easily deployed, and run on Nebula. We also explore a range of failure scenarios, a common occurrence in volunteer systems, and show our system is robust to arbitrary failures of both hosted compute and data nodes that may occur anywhere within the system.

We compare results to two systems modeled on volunteer platforms that have been proposed in the literature, one with

978-1-4799-3766-0/14 $31.00 © 2014 IEEE
DOI 10.1109/IC2E.2014.34

a centralized filesystem [1], and one further tuned for MapReduce by allowing distributed intermediate data [5], running on our compute and data platforms. We show that the Nebula approach can greatly outperform both baselines on our testbed (by 580% and 200% respectively for a common MapReduce application with 1GB of input data) and exhibits good scaling and fault tolerance properties.

II. Nebula

In this section we describe the design of Nebula, a location and context-aware distributed cloud infrastructure that is built using voluntary edge resources. Both volunteer nodes and input data could be spread across the world, and Nebula will take advantage of this geographic spread by locating nearby computational resources.

A. Nebula Design Goals

Nebula has been designed with the following goals in mind:

- *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, in addition to an efficient computational platform, 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 pre-loaded to a central location.
- *Location-aware resource management*: To enable efficient execution of distributed data-intensive applications, Nebula must consider network bandwidth along with computation capabilities of resources in the volunteer platform. As a result, resource management decisions must optimize on 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.
- *Sandboxed 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.
- *Fault tolerance:* Nebula must ensure fault tolerant execution of applications in the presence of node churn and transient network failures that are typical in a volunteer environment.

B. Nebula System Architecture

Figure 1 shows the Nebula system architecture. Nebula consists of volunteer nodes that donate their computation and storage resources, along with a set of global and application-specific services that are hosted on dedicated, stable nodes. These resources and services together constitute four major components in Nebula (described in more detail in Section III):

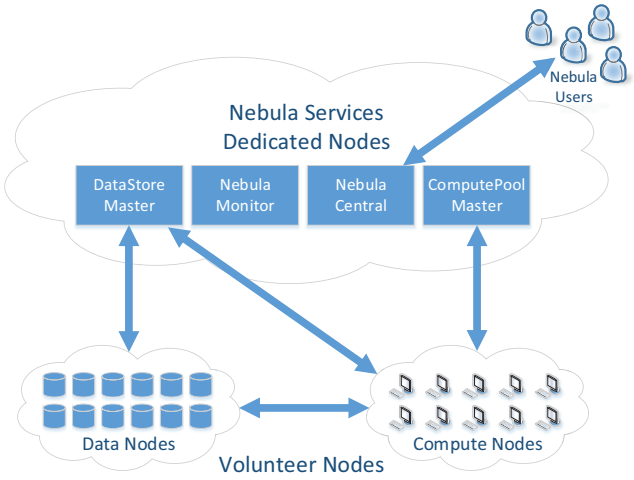


Fig. 1. Nebula system architecture.

- *Nebula Central*: Nebula Central is the front-end for the Nebula eco-system. It provides a simple, easy-to-use Web-based portal that allows volunteers to join the system, 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 processing in Nebula. Each DataStore consists of volunteer data nodes that store the actual data, and a DataStore Master that maintains the storage system metadata and makes data placement decisions.
- *ComputePool*: The ComputePool provides per-application computation resources through a set of volunteer compute nodes. Code execution on a compute node is carried out inside a Google Chrome Web browser-based Native Client (NaCl) sandbox [21]. 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 computation requirements and data location.
- *Nebula Monitor:* The Nebula Monitor does performance monitoring of volunteer 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.

These components work with each other to enable the execution of data-intensive applications on the Nebula platform. Each volunteer node can choose to participate as a data node or a compute node depending on whether it donates its storage, compute resources, or both. The volunteer nodes are multiplexed among different applications, so each compute

(or data) node can be part of multiple ComputePools (or DataStores). The centralized components only provide control and monitoring information, and all data flow and work flow happens directly between the volunteer nodes in the system. As a result, these components do not become bottlenecks in the execution path.

C. Nebula Applications

A Nebula application may consist of a number of *jobs*. A job contains the code to carry out a specific computation. For example, a MapReduce application contains a map and reduce job in the Nebula parlance. An application execution is referred to 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. In this work, we assume the data has already been stored into Nebula and that there are many more files than tasks, thus there is no need to further decompose the input files (though they may require aggregation). Future work will explore how external data can be inserted into Nebula and either aggregated or decomposed as needed. 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 consists of multiple *tasks* (typically identical in structure), which can be executed in parallel on multiple compute nodes. Each task is associated with the job executable code and a data partition of the input dataset, upon which the code is executed. The executable code is Native Client code (a set of .nexe 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.

III. Nebula System Components

A. Nebula Central

Nebula Central is the front-end for the Nebula eco-system. 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 volunteers can join to contribute compute and storage resources to Nebula, and application writers (or users) can upload their applications and initiate execution. This API is used to load the application executables, input files, and DataStore parameters for the various application files (Table I). Nebula Central uses these parameters 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 that handle the data placement and job execution for the application.

The input data management depends on the desired behavior of the DataStore as specified by the application. For example, it may store multiple copies of the data so that each online DataStore node gets a share proportional to the inverse of that node's average upload bandwidth. This behavior allows computation to have good locality to the data sources. Similarly, the application can customize the scheduling behavior of the ComputePool. Some of this customization occurs via parameterization and in other cases by component replacement (envisioned in the future). We presently support a set of default policies that are shown to perform well in our experiments. Nebula Central also supports the concurrent execution of multiple applications and provides tools to monitor progress via the generation of user-friendly dynamic graphs and maps.

TABLE I. Nebula Central job 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 filenames to use as source inputs for the jobs.
output file prefix	Application	The filename prefix to use for outputs of jobs.
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.
job scheduler	Application	The scheduler to be used by the ComputePool Master.
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	Various settings for a task including timeouts, minimum bandwidths, replication settings, and failure tolerances.

B. DataStore

The DataStore is designed to provide a simple storage service to support data processing on Nebula. In volunteer computing, nodes are typically interconnected by WAN and bandwidth is usually low. Optimizing data transfer time is crucial to efficient data processing in this case. Nebula Central will ultimately decide how to partition or share resources across multiple applications to meet their requirements.

Data in a DataStore is stored and retrieved in units of files. Files are organized using namespaces. 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.

1) Data Node: The data nodes are volunteer nodes that donate storage space and store application data files. A data node is implemented as a light-weight Web server. The data nodes support two basic operations—`store` and `retrieve`—that are exposed to clients to store and retrieve files to and from the DataStore.

It is the responsibility of the data node to update the entries corresponding to a file in the associated DataStore Master

TABLE II. DATASTORE MASTER OPERATIONS

Operation	Parameters	Description
`get_data_nodes_to_store`	None	Returns an ordered list of data nodes for a client to store data
`get_data_nodes_to_retrieve`	filename, namespace	Returns an ordered list of data nodes for a client to retrieve data
`set`	filename, namespace, nodeid, filesize	Sets a new entry for a file
`ping`	nodeid, isonline	Reports a data node as being online
`found`	nodeid	Reports another node as online (for P2P communication)

whenever a new file is uploaded. Each data node also sends heartbeats to the DataStore Master(s) to register itself and to let it know that it's online.

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. All the intelligence is part of the DataStore Master and is transparent to the clients. The only overhead on the client is to be able to detect the failures of data nodes and fall back to other nodes provided by the DataStore Master (discussed below).

2) DataStore Master: The DataStore Master keeps track of data nodes that are online and file metadata. The DataStore Master supports a set of operations to manage the data nodes and to carry out effective data placement decisions (Table II). Some operations are invoked by DataStore clients before storing or retrieving data to/from the DataStore. Others are invoked by the data nodes and are used to exchange metadata and health information.

Load balancing and locality awareness: The DataStore Master can achieve better performance via load balancing and locality awareness by appropriately ordering the lists returned to the clients via the `get_data_nodes_to_*` operations. To achieve better load balancing, the list of nodes can be randomized or ordered by their load (or available storage capacity). On the other hand, locality awareness can be achieved by ordering data nodes based on their bandwidth, latency or physical distance from the requesting client. For 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 data node at the head of the list.

Fault tolerance: The DataStore Master achieves fault tolerance through replication of data. A user can specify replication parameters (number of replicas) for the application data, and the Master then ensures that the requested number of replicas are maintained for each file in the DataStore. It keeps track of online data nodes and actively tries to replicate data if the number of replicas falls below a threshold. The list of data nodes returned to the client also helps provide fault tolerance in the event that the preferred node in the list fails. Since such failures are expected, providing a list reduces the number of roundtrip calls to the Master that a client needs to make and removes the need for maintaining state at the DataStore Master.

C. ComputePool

The ComputePool provides computational resources to Nebula applications. 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.

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 NaCl sandbox [21] provided by the Google Chrome browser. The compute node will be provided a Web page for each task execution that contains Javascript code with embedded native code for efficiency. The use of NaCl gives us several advantages, the largest of which is that a volunteer node is protected from malicious code while performing at near native-speed. *All it takes to join Nebula for a Chrome user is to enable the NaCl plugin and point the browser to Nebula Central*. 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 and processing while streaming the data.

Fault tolerance: The compute node can handle a variety of failure types including issues starting the NaCl subsystem, missing or corrupt data files, and slow file transfers. The system uses heartbeats to ensure the system is running properly. If the heartbeats are missed, the compute node reports the task has failed and attempts to run a new task. In most cases, the task will just be reattempted, either by the same node or another node. However, if multiple nodes have reported a single data node for slowness, or if the input data no longer exists, the ComputePool Master will try to regenerate the input files if it was produced by an earlier task execution.

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.

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[1]. The scheduler uses monitoring information to decide which tasks should be run and where. The scheduling decisions can be as simple as a random scheduler, to the more complex adaptive MapReduce scheduler, which attempts to approximate network and computation abilities and needs to complete a task in minimal time despite nodes going up and down during execution.

As an example, we present the *locality-aware scheduling algorithm*, LA, that will be used for MapReduce later:

1) Get updates about the current state of the network.

[1]We envision that schedulers can be defined on a per-framework basis, e.g., for MapReduce, and shared by many applications.

2) Estimate remaining time to completion for each *running* task. If the estimate has been exceeded, adjust the projected running time to account for the inaccuracy.
3) For each *not running* task, create estimates for each (node, task) pair. This estimate is based on the transfer time for task inputs and results to the DataStore and execution time.
4) For each (node, task) pair, we estimate its time to finish based on the remaining time for any running task on a node plus the estimate of running the task on that node. We then order these pairs in a priority queue using this estimated finishing time.
5) For each task, we select the single best node 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.

Load balancing and locality awareness: To achieve load balancing, the compute scheduler can take into account the CPU speeds and current loads on different compute nodes to assign them tasks. To achieve locality awareness the compute scheduler 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 location, and assigns tasks from this list when the compute node requests additional work. This location-based task allocation mechanism, along with the locality awareness support in the DataStore, can reduce network overhead substantially.

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.

D. Nebula Monitor

The Nebula Monitor does performance aggregation of compute clients and data nodes. It accepts queries from the other Nebula Services 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. At the moment, 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 metrics like bandwidth, latency, etc. between the compute nodes and Nebula Central that can be used as defaults for unknown information.

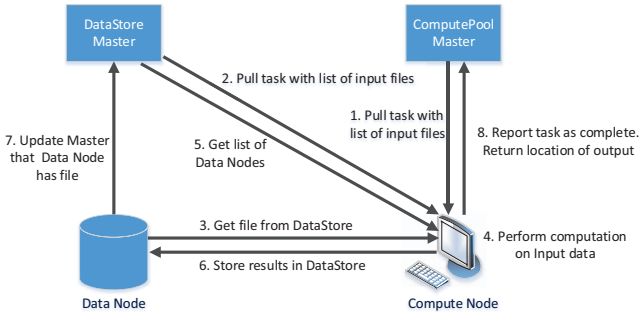


Fig. 2. Control and data flow and steps involved in executing a task on the Nebula infrastructure.

E. Task Execution in Nebula

Having described the different components of Nebula, we now put them all together and describe how a task is executed in Nebula. 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 tasks and start executing them. Figure 2 shows the various steps involved in the execution of a task. The compute node contacts ComputePool Master periodically and asks for tasks. The scheduler would assign tasks to the compute node based on the scheduling policy. The compute node would then download the application code, as well as the input data from the DataStore Nodes. The computation starts in NaCl as soon as these downloads are completed. Once computation is performed, the outputs are then uploaded back to the DataStore. Finally, DataStore bandwidths between the compute node and the DataStore platforms as well as the location of the output files are provided to the ComputePool Master.

The size and composition of the ComputePool and DataStore are application-specific. In addition, how global Nebula resources are allocated across applications is controlled by a policy implemented at Nebula Central. These issues are outside the scope of the current paper but discussed in the Future Work section.

IV. MapReduce on Nebula

To illustrate the efficacy of using Nebula for distributed data-intensive computing, we have implemented a MapReduce application framework on top of Nebula. MapReduce [6] is a popular programming paradigm for large-scale data processing on a cluster of computers, and provides a simple programming framework for a large number of data-intensive computing applications. For this reason, we have selected MapReduce as our first live Nebula application framework as a test of our approach. We note that our Nebula MapReduce framework is not based on existing MapReduce implementations such as Hadoop/HDFS; rather it is implemented completely on top of the Nebula infrastructure and utilizes services such as DataStore and ComputePool for all storage and compute needs.

A MapReduce Nebula application consists of two jobs, map and reduce. The reduce is dependent on the map job, so it cannot execute until the map job completes. Writing applications for the Nebula framework involves a few simple steps. Map and reduce jobs are written in C++ and compiled

against the specialized compilers provided by the NaCl SDK. The output of the compilation phase is a set of .nexe files which can be uploaded to Nebula Central for distribution via its Web interface. The input data of a MapReduce application needs to be pushed to the DataStore first and all these inputs need to 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 tasks, number of reduce tasks, namespace, and a list of inputs.

Once a MapReduce application has been instantiated, the ComputePool Master can start assigning tasks to compute nodes. Each map task obtains its input data from the DataStore, performs the map function on it, and the result list is partitioned into as many output files as there are reduce tasks. The output files are uploaded to the DataStore at the end of each map task. A reduce task downloads the map outputs for its partition, carries out the reduce operation, and uploads the output back to the DataStore.

A MapReduce scheduler was designed to optimize MapReduce task execution and data movement and is used by the ComputePool and DataStore Master 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. This is the locality-aware scheduler of Section III-C2.

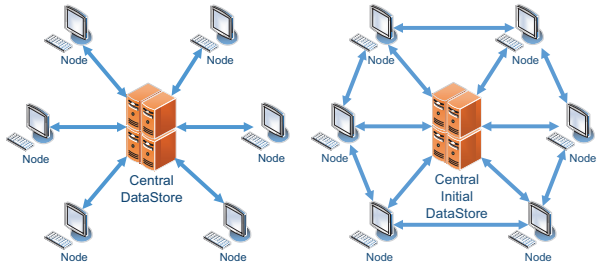
V. Evaluation

To evaluate the performance of Nebula, and specifically MapReduce on Nebula, we have configured an experimental setup on PlanetLab [3], where we have deployed Nebula and carried out a set of experiments. We also emulate several existing volunteer computing models on PlanetLab and treat them as a baseline for our evaluations.

A. Experimental Setup

For our experiments, we have set up 52 nodes on PlanetLab [3] 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 256Kbps to 32Mbps. All the dedicated Nebula services (Nebula Central, DataStore and ComputePool Masters, and Nebula Monitor) are hosted on a single machine with Dual Core Pentium E2200 @ 2.4GHz, 4GB RAM, 150GB Hard Drive, running Ubuntu 12.04 Linux. It also hosts MySQL databases to support Nebula Central and ComputePool Masters (2.2GB), and Redis databases (800MB) to support Nebula Central and DataStore Masters.

We perform experiments using the Nebula MapReduce Wordcount and InvertedIndex 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 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.



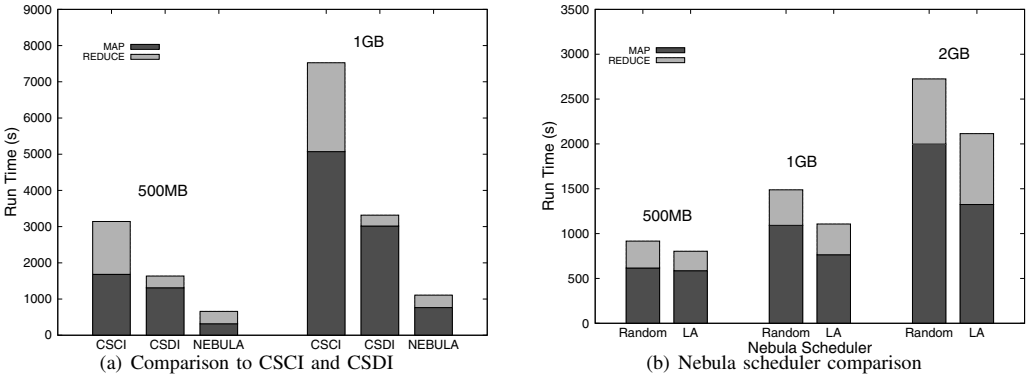
(a) CSCI (b) CSDI

Fig. 3. Data flow in CSCI and CSDI.

For comparison, we emulate two alternate volunteer computing models on top of the Nebula infrastructure: *Central Source Central Intermediate Data (CSCI)* and *Central Source Distributed Intermediate Data (CSDI)*. These models correspond to the data models supported by BOINC [1] and a MapReduce-tuned BOINC version [5] 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 output) is different. In CSCI, intermediate data is stored centrally 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 data flow for the CSCI and CSDI model are illustrated in Figure 3.

In the Nebula model, the input data is already randomly distributed data as would be the case in the applications we are targeting (see Section I for examples). Given this data placement, Nebula tries to select the best nodes for computation to minimize the overall execution time. We compare CSCI and CSDI against Nebula MapReduce using both a random scheduler (Nebula-Random) and the locality-aware scheduler (Nebula-LA) presented earlier (Section III-C2).

In all 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. The input data is placed on a fixed set of 8 randomly selected data nodes with a replication factor of 2. Intermediate and output data can be placed on any available data node, and is not replicated unless specified.



(a) Comparison to CSCI and CSDI (b) Nebula scheduler comparison

Fig. 4. MapReduce comparison across different environments.

B. Performance Comparison

The first set of experiments directly compare different approaches in a volunteer environment. The MapReduce Wordcount application was run multiple times on each of the different environments with different size datasets (500MB, 1GB, and 2GB). All experiments used 300 map tasks each, along with 80, 160, and 320 reduce tasks for the 500MB, 1GB, and 2GB input sizes respectively. 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.

Figure 4(a) 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 1GB, Nebula continues to perform better (by 580% and 200% vs. CSCI and CSDI respectively), and the larger data size shows the additional benefit of locality-awareness by selecting compute nodes based on performance estimation.

To see the impact of choice of scheduler, Figure 4(b) compares the locality-aware (LA) scheduler to a Random scheduler within Nebula, which randomly assigns tasks to compute nodes. We find that the LA scheduler outperforms the Random scheduler by 16-34% 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.

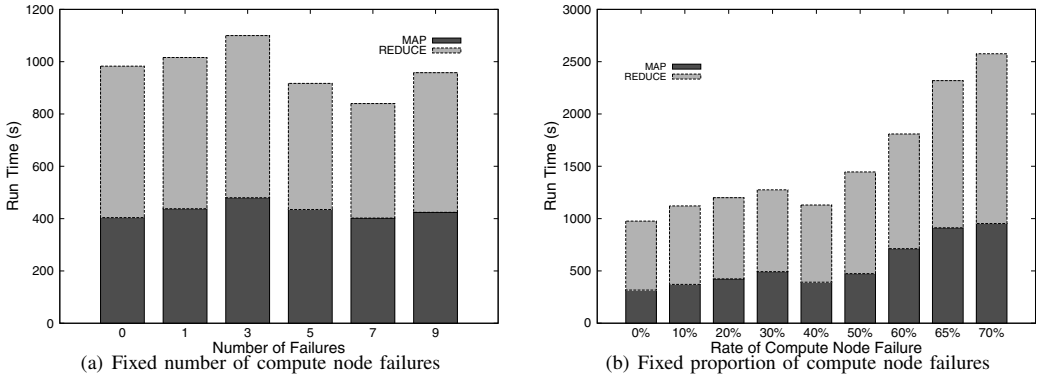
C. Fault Tolerance

In the next set of experiments, we induce crash failures in both the compute and data nodes during execution. As discussed in Section III, 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 500MB experiment with Nebula-LA (Figures 4(a) and 4(b)), 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 500MB 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.

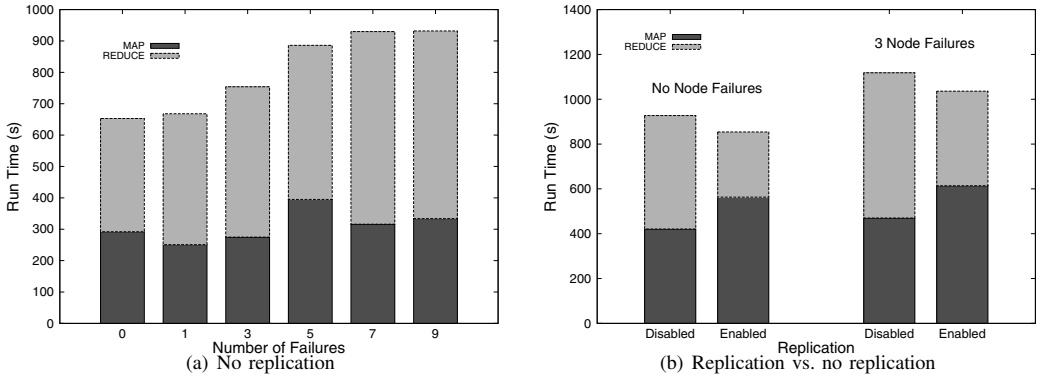
1) Compute node failures: We first show the impact of compute node failures. In these experiments, we do not fail the data nodes, and do not use data replication for any of the data in the system. In the first experiment, we kill a subset of compute nodes part-way (50%) through the execution of the map 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. Figure 5(a) 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. Figure 5(b) shows the results of inducing these failures. In this case, we see that beyond a certain rate of failures (over 50%), the runtime deteriorates. At this point there



(a) Fixed number of compute node failures (b) Fixed proportion of compute node failures

Fig. 5. Performance of Nebula in the presence of compute node failures.



(a) No replication (b) Replication vs. no replication

Fig. 6. Performance of Nebula in the presence of data node failures.

are so many compute failures that the system starts becoming compute-limited, and has to re-execute many unfinished tasks.

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 part-way (50%) 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. Figure 6(a) 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 Figure 5(a).

Finally, we show the benefit of using data replication for intermediate data as well. In this case, we enable replication which creates multiple copies of intermediate data, so that the system can handle data node failures without the need for re-executing tasks. Figure 6(b) compares the system performance with replication enabled against no replication for two scenarios: one without data node failures, and one with 3 node failures. For the 3 node failure case, we kill 3 data nodes 50% through the execution of the map phase, similar to the first experiment. We see that the total runtime is reduced for the replication case. This is because even though replication adds overhead resulting in higher map times, the reduce times are lower, since the system does not need to re-execute tasks corresponding to lost data. In fact, the runtime is lower even for the no failure case, because with more replicas available, the ComputePool Master has more choices to assign tasks to compute nodes in a locality-aware manner.

D. Scalability

In these experiments, we present results which show that Nebula is able to scale up with the number of nodes and data size. 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. Figure 7 shows the performance of Neb-

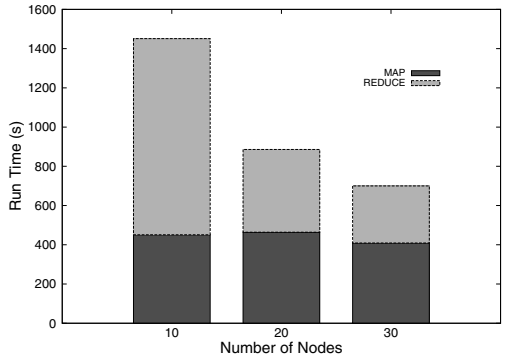


Fig. 7. Node Scalability of Nebula.

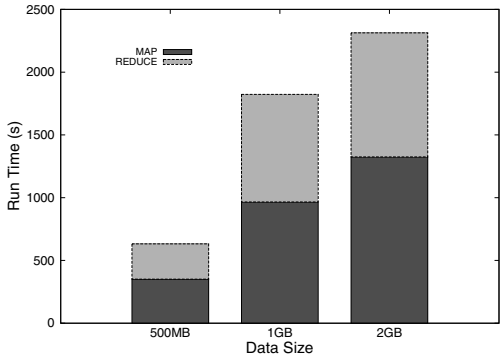


Fig. 8. Data Scalability of Nebula.

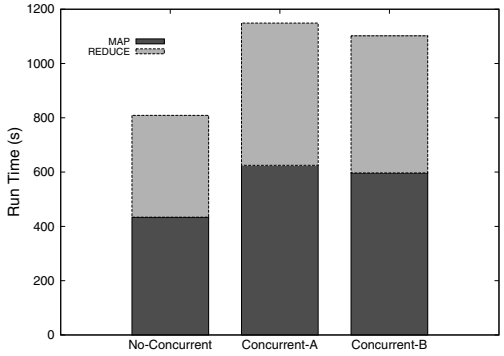


Fig. 9. Concurrent Nebula applications.

ula as we increase the number of compute and data nodes in the system from 10-30 each (i.e., 30 refers to 30 compute nodes and 30 data nodes). All these experiments use an input data size of 250MB, along with 250 map and 80 reduce tasks. The input data is still 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 with increasing system size. 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 better compute resources w.r.t. the location of the intermediate data. We hope to fully validate this encouraging preliminary scale result on a much larger edge system in the future.

To see the scalability of Nebula in the presence of large intermediate data as well, we present results for a second 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. Figure 8 shows the performance of Nebula as it creates an inverted index from 500MB, 1GB, and 2GB text files. As shown in Figure 4(b), the total runtime increases as we increase data size. Though the InvertedIndex causes expansion of the input data, the results are similar to the previous results for Wordcount (Figure 4(b)) which provides high reduction of input data. We see that the system still scales with InvertedIndex as with Wordcount as the problem size expands.

E. Concurrent Applications

Lastly, we show that Nebula is able to run concurrent MapReduce applications. Figure 9 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 250MB, 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. Nebula currently multiplexes its resources equally among concurrent applications, and more sophisticated cross-application resource management policies (e.g., proportional-share or priority-based) are part of future work discussed later.

VI. Related Work

Nebula is related to projects in a number of different areas. Volunteer edge computing and data sharing systems are best exemplified by Grid and peer-to-peer systems including, Kazaa [17], BitTorrent [4], Globus [8], Condor [13], BOINC [1], and @home projects [15]. 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. Cloud4Home [10] is focused 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. Cloudlets [18] is a localized cloud designed for latency-sensitive mobile offloading though it is based on heavyweight virtualization technologies.

There are a number of relevant distributed MapReduce projects in the literature [12], [14], [9]. Moon [12] is focused on voluntary resources but not in a wide-area setting. Hierarchical MapReduce [14] is concerned with compute-intensive

MapReduce applications and how to apply multiple distributed clusters to them, but uses clusters and not edge resources. Our recent work [9] is focused more on cross-phase MapReduce optimization, albeit in a wide-area setting. Estimating network paths and forecasting future network conditions are addressed by projects such as NWS [20]. We have used simple active probing techniques and network heuristics for prototyping and evaluation of network paths in our Nebula Monitor. Existing tools [16], [7], [11] would give us a more accurate view of the network as a whole.

VII. Conclusion and Future Work

We presented the design of Nebula, an edge-based cloud platform that enables distributed in-situ data-intensive computing. The Nebula architectural components were described along with abstractions for data storage, DataStore, and computation, ComputePool. A working Nebula prototype running across edge volunteers on the PlanetLab testbed was described. An evaluation of MapReduce on Nebula was performed and compared against other edge-based volunteer systems. The locality-aware scheduling of computation and placement of data enabled Nebula MapReduce to significantly outperform these systems. In addition, we showed that Nebula is highly robust to both transient and crash failures. Future work includes expanding the range of data-intensive applications and frameworks ported to Nebula and validation of our initial findings on a much larger scale. We also plan on first-class support for resource partitioning across frameworks and applications running across shared Nebula resources. Lastly, we plan on expanding the range of DataStore features to include the injection of external data and techniques for both aggregation and decomposition across distributed resources.

Acknowledgment

The authors would like to acknowledge NSF Grant: NSF-CSR 1162405, which supported this research.

References

[1] D. P. Anderson. BOINC: A System for Public-Resource Compting and Storage. In *Proceedings of the 5th ACM/IEEE International Workshop on Grid Computing*, 2004.

[2] A. Chandra and J. Weissman. Nebulas: Using Distributed Voluntary Resources to Build Clouds. In *HotCloud'09: Workshop on Hot topics in cloud computing*, June 2009.

[3] 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*, 33(3):3–12, July 2003.

[4] B. Cohen. Incentives build robustness in BitTorrent. In *Proceedings of the First Workshop on the Economics of Peer-to-Peer Systems*, June 2003.

[5] F. Costa, L. Silva, and M. Dahlin. Volunteer Cloud Computing: MapReduce over the Internet. In *Fifth Workshop on Desktop Grids and Volunteer Computing Systems (PCGRID 2011)*, May 2011.

[6] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Sixth Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[7] A. Downey. Using pathchar to estimate Internet link characteristics. In *In Proceedings of ACM SIGCOMM*, pages 241–250, 1999.

[8] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. In *Proceedings of the Global Grid Forum*, June 2002.

[9] B. Heintz, C. Wang, A. Chandra, and J. B. Weissman. Cross-Phase Optimization in MapReduce. In *Proceedings of the IEEE International Conference on Cloud Engineering*, IC2E'12, 2013.

[10] S. Kannan, A. Gavrilovska, and K. Schwan. Cloud4Home – Enhancing Data Services with @Home Clouds. *International Conference on Distributed Computing Systems*, pages 539–548, 2011.

[11] K. Lai and M. Baker. Measuring link bandwidths using a deterministic model of packet delay. In *in Proceedings of ACM SIGCOMM*, pages 283–294, 2000.

[12] H. Lin, X. Ma, J. Archuleta, W.-c. Feng, M. Gardner, and Z. Zhang. MOON: MapReduce On Opportunistic eNvironments. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, 2010.

[13] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor - a hunter of idle workstations. In *ICDCS*, pages 104–111, 1988.

[14] Y. Luo and B. Plale. Hierarchical MapReduce Programming Model and Scheduling Algorithms. In *12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, May 2012.

[15] D. Molnar. The SETI@Home problem. *ACM Crossroads*, Sept. 2000.

[16] A. Pasztor and D. Veitch. Active Probing using Packet Quartets. In *In ACM SIGCOMM Internet Measurement Workshop*, pages 293–305, 2002.

[17] S. Saroiu, K. P. Gummadi, R. J. Dunn, S. D. Gribble, and H. M. Levy. An Analysis of Internet Content Delivery Systems. In *Proc. of Symposium on Operating Systems Design and Implementation*, 2002.

[18] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The Case for VM-Based Cloudlets in Mobile Computing. *IEEE Pervasive Computing*, 8(4):14–23, October 2009.

[19] J. B. Weissman, P. Sundarrajan, A. Gupta, M. Ryden, R. Nair, and A. Chandra. Early Experience with the Distributed Nebula Cloud. In *Proceedings of the fourth international workshop on Data-intensive distributed computing*, DIDC '11, pages 17–26, 2011.

[20] R. Wolski, N. T. Spring, and J. Hayes. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *Journal of Future Generation Computing Systems*, 1999.

[21] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullaga. Native client: a sandbox for portable, untrusted, x86 native code. In *Proceedings of IEEE Security and Privacy*, 2009.