Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 Keller Hall
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 13-007

Nebula: Data Intensive Computing over Widely Distributed Voluntary Resources

Mathew Ryden, Abhishek Chandra, and Jon Weissman

March 14, 2013

Nebula: Data Intensive Computing over Widely Distributed Voluntary Resources

Mathew Ryden, Abhishek Chandra and Jon Weissman
Computer Science and Engineering
University of Minnesota
Minneapolis, MN 55455
{mathew, 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 emulated BOINC.

1. 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. The Nebula project has been exploring a different part of the data-intensive landscape. 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 edge resources for both computation and data storage to address the first two aspects. If cost is an issue, then we advocate the use of volunteer resources, the benefits of which are well-known to the HPDC community. Today, the edge is even more attractive 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 less of an issue, then Nebulas can use monetized resources across CDNs or even ISPs, provided these were equipped to offer computational services. In this paper, we explore 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 [5], 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.

To give a flavor of why the edge may be suitable for data-intensive computation, consider the following examples. A user wishes to analyze a set of blogs located across the Internet to identify opinions about the current election. A different user wants to analyze video feeds taken from geographically separated cameras across a set of airports looking for suspicious activity. These examples have the following characteristics making the edge attractive: each data item can be processed independently in-situ (or close to the data location) and the data processing yields significant data compression. In the first case, only a very small subset of a blog is returned (any mention of politics) and in the latter case only portions of the video that contain potentially suspicious activity for further processing.

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 [4] nodes across Europe. An early version of Nebula was described in our prior work [3, 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 volunteer platforms that have been proposed in the literature, standard BOINC [1], and a version of BOINC further tuned for MapReduce [6]. We show that the Nebula approach can greatly outperform both baselines on our testbed by a number of locality-based optimizations and ex-

hibits good scaling properties. For instance, on a common MapReduce application with 1GB of input data, the Nebula system performs 580% and 200% faster than emulated BOINC and MapReduce-tuned BOINC respectively.

2. NEBULA

In this section we describe Nebula, a location and context-aware distributed cloud infrastructure that is built using voluntary edge resources. One of the motivations for Nebula is that standard data-intensive frameworks perform poorly in widely distributed data scenarios. In prior work, we have shown an instance of MapReduce, Hadoop, performs very poorly due to the large amount of wide-area data flow [2]. Nebula is designed for wide-area distributed data-intensive computing. It provides two basic services: data storage and task computation. Each volunteer node can choose to donate storage space, computational resources or both. Volunteers can choose to join and leave Nebula at any time. 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. Nebula is an evolving project and we present the current state of the project in this paper.

2.1 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.

- *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.

- *Extensible programming model:* Application programmers must be able to write their code in a variety of languages including native languages such as C/C++ or traditional scripting languages such as Python/Perl/Tcl, and Nebula must provide hooks by which such applications can be executed in a parallel manner.

- *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.

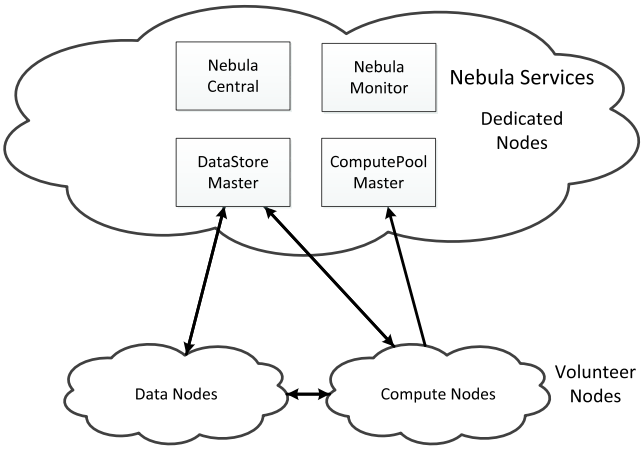


Figure 1: Nebula system architecture.

2.2 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 3):

- *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 keeps track of 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.

It is important to stress that our work is orthogonal to and does supplant other abstractions for data storage (e.g. BitTorrent) or computation (e.g. PlanetLab slice). Our abstractions are defined at a more general level to enable application-specific tuning and customization. Thus, a specific version of a DataStore *could* employ a BitTorrent-like protocol underneath if the application required it. 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). We must note that the centralized components (Nebula Central, DataStore and ComputePool Masters, and the Nebula Monitor) 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, though they may be scaled up (e.g., via replication or partitioning) to support large number of users and nodes in the system. We also require the centralized components to be hosted on stable, dedicated nodes, as they maintain metadata and control information for the system. While we refer to them as single logical components for ease of exposition, they may be partitioned or replicated for performance and fault tolerance.

2.3 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.

3. NEBULA SYSTEM COMPONENTS

3.1 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 1). 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. It then provides the necessary application components to the DataStore and ComputePool Masters, such as the application executable code.

How the input data is managed 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. In addition, if there are more input files than tasks, it may aggregate input files together to minimize the number of DataStore nodes accessed by the task. This latter behavior matches the distributed data-intensive applications we are targeting. The opposite case, decomposing an already existing large file, is planned as future work. Collectively, these behaviors allow 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 data-intensive applications and provides tools to monitor progress via the generation of user-friendly dynamic graphs and maps.

Nebula Central also provides a well-known point of entry for volunteer nodes. A volunteer node joins the system by registering with Nebula Central. A data node downloads a generic piece of code that can carry out data node functions such as storing/retrieving files, and talking to DataStore Masters and clients. 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 data nodes volunteers will also use a browser to download their infrastructure via signed Java applets or similar technology. The volunteer node is then assigned to one or more DataStore and/or ComputePools,

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.

Table 1: Nebula Central job inputs

and interacts directly with their respective Master nodes to carry out data storage and/or computation 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 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. Each application owns a DataStore that it may configure to support desired performance, capacity, or reliability constraints. Volunteer data storage nodes may be multiplexed across potentially different DataStores. 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. 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 are volunteer nodes that donate storage space and store application data files. A data node is implemented as a light-weight Web server that stores and serves files. The data nodes support two basic operations that are exposed to clients to store and retrieve files. These operations are:

- `store(filename, namespace, file_contents)`: This operation stores a file into the DataStore. Once the file is written to disk and saved successfully, the DataStore Master is updated with a new file record.
- `retrieve(filename, namespace)`: This operation retrieves the file specified by the filename and namespace combination from the DataStore.

It is the responsibility of the data node to update the entries corresponding to a file in the associated DataStore Master 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. These operations can also be used to place data in an optimal fashion before the computation even begins. 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).

3.2.2 DataStore Master

The DataStore Master keeps track of several items: data nodes that are online, file metadata and bandwidth information between nodes. It serves as an entry point of 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. The set of operations are:

- `get_data_nodes_to_store()`: This operation returns an ordered list of data nodes a client can use to store data. A client first makes a call to this operation before uploading data to the DataStore, and tries to upload the file to the first node in the returned list. The client falls back on the next node in the list if the first node fails. The returned list is not static, and can depend on various factors such as the availability and load of data nodes, location of the client, etc.
- `get_data_nodes_to_retrieve(filename, namespace)`: This operation returns an ordered list of data nodes (URLs) for a particular file that could be located on multiple nodes. The client will then try to retrieve the file from the returned URLs in order. Similar to the above operation, the list returned here is also dynamic and can vary based on the source of the request and the data nodes that are currently online.
- `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. The filesize is an example of file metadata that the DataStore Master keeps track of. It is currently used to estimate the file transfer.
- `ping(nodeid, isonline)`: This operation reports that a

data node is online, and is needed to ensure that a data node is able to contact the DataStore Master. The data nodes periodically ping the DataStore Master to notify their status as online. Data nodes that do not check in after a period of time are automatically 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.

• `found(nodeid)`: This operation marks a node which previously performed a `ping` as online. This is done to ensure peer-to-peer communication between volunteer nodes. e.g., between a compute node and a data node. This operation is carried out by a data node (say, D_1) after it calls `ping`: D_1 is returned the nodeid (say, D_2) When a ping is received, it returns the URL for another node which has recently performed a `ping`. The goal is to check if D_2 can be contacted directly by D_1 within the peer network, and if so, then the `found` operation is invoked to mark the node D_2 as actually functional. When this URL is followed, it gets information for a follow-up **found()** command to mark The reason for the two step `ping-found` process is to ensure that nodes have working two-way communication not only with the master but with other volunteer nodes as well. and to perform other health-related checks. For example, this tests if a node can receive an incoming connection. The URL returned from the **ping()** command when accessed runs a few checks to ensure it is able to create and remove files in all of the necessary places as well as the implicit requirement to be able to both open and accept socket connections. These checks are included as hosts sometime find themselves unable to perform some of them, although all volunteer nodes were able to perform all of them sometime.

These operations are implemented over HTTP, so parameters like the client's 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 data is transfered between two nodes in the system. Since this data is not available for newly joined data nodes, we insert them in a random position in the returned node list. This is to ensure that new nodes do not get starved, and bandwidth/latency 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.

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. As an extreme 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 through the `ping` and `found` operations, 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.

Operation	calls/sec	ms/call
`get_data_nodes_to_retrieve`	19.3 ± 4.2	42.5 ± 5.7
`get_data_nodes_to_store`	45.9 ± 3.0	48.3 ± 5.7
`set`	58.7 ± 4.3	43.8 ± 5.7

Table 2: DataStore Master microbenchmarks

Microbenchmarks: The DataStore Master is a critical piece of infrastructure which may be called hundreds or thousands of times during a Compute task. To ensure its performance is meeting demands, we track the performance of the calls from data nodes as the system is running. Measurements from when the system was under load are shown in Table 3.2.2. The data shown is only for measurements when the system was under at least 1 call per second of each of the operations to see how the calls interact with eachother. This data is collected on the DataStore Master from the time it begins to handle the call to when it sends the response. Actual average wait time will be larger due to network and machine delays.

3.3 ComputePool

The ComputePool provides computational resources to Nebula applications. It is the computational analogue to the DataStore. As such, 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 [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. 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. Secondly, it makes it easy for users to join Nebula, because NaCl comes with Google Chrome - a popular, widely available Web browser. *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. Downloading the application code and transferring context from the encapsulating Javascript layer to the NaCl plugin takes some time, but this is a one time cost. Once downloaded, the

application can be cached on the client side. The sandbox adds little overhead to the application, and execution times inside and outside the sandbox are comparable [21]. 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. Finally, from an application developer's perspective, NaCl provides the flexibility to code in natively-compiled languages such as C/C++, with the only requirement that the code be compiled with the NaCl compiler.

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 a ComputePool Master URL. The 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 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. It then 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 failure types. The NaCl files can fail to load, the NaCl subsystem can fail to launch the executable, the NaCl module can crash after starting, files can be unavailable, files can be corrupt, and file transfers can be slow. The NaCl code sends heartbeat messages to the javascript layer, which cancels the task if it doesn't 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 just 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 ask the DataStore Master to move the data to another node. If the input data doesn't exist in the DataStore, Nebula will attempt to 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[1]. To support the scheduler, the ComputePool Master implements a dashboard which collects per-node execution statistics and the Nebula Monitor tracks inter-node link bandwidths. The scheduler uses monitoring information to decide which tasks should be run and where. The complexity of scheduling decisions can be as simple as a random scheduler, to more complex as our 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. The scheduler runs periodically updating node to task mappings and when nodes check in for work they are allocated the assigned task(s).

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

1. Get updates from the ComputePool and DataStore Masters and the Nebula Monitor about the current state of the system.

2. Estimate remaining 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 unallocated task, create estimates for each (node, task) pair. This estimate is based on the download time for all of the task inputs from the DataStore, the time to execute the task, and 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 single 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.

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, it

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

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.

3.4 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 measures certain metrics like bandwidth, latency, etc. between the compute nodes and Nebula Central that can be used as defaults if we do not know link-to-link data. 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 Task Execution in Nebula

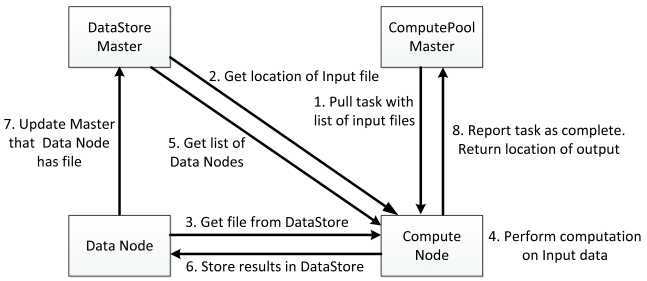


Figure 2: Control and data flow and steps involved in executing a task on the Nebula infrastructure.

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.

4. 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 [7] 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.

4.1 Creating Nebula MapReduce Applications

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.

4.2 Example Application: Wordcount

As a way of evaluating MapReduce on Nebula, we have implemented a Wordcount application which counts the total frequency of each word appearing in a number of ebooks. Figures 3 and 4 show the C++ code snippets for the Map and Reduce classes.

Each map task obtains its input data from the DataStore, performs Wordcount on it, and the result is a list of key-value pair with key being the word and value being the frequency of the word in the inputs. The list is then partitioned into as many output files as there are reduce tasks by using a

```
class Map : public Nebula::MapNebulaTask {
  void map(string data) {
    string word;
    wordstream wss(data);

    while(wss >> word) {
      emit(word, "1");
    }
  }

  void handleAllDownloadsFinished() {
    for(int i = 0; i < reduceCount; i++)
      uploadHelper(i, getTaskOutputInString(i));
  }
}
```

Figure 3: Wordcount Map Code

```
class Reduce : public Nebula::ReduceNebulaTask {
  map<string, vector<string> > wordcounts;

  void handleDownloadData(string data) {
    JSONValue *value = JSON::Parse(ws);
    JSONArray root = value->AsArray();

    for(auto i = root.begin(); i != root.end(); i++) {
      JSONArray entry = (*i)->AsArray();
      string word = entry[0];
      JSONArray in = entry[1]->AsArray();
      for(auto ii = in.begin(); ii != in.end(); ii++) {
       wordcounts[word].push_back(*ii);
      }
    }
    delete value;
  }

  void handleAllDownloadsFinished() {
    auto end = wordcounts.end();
    for(auto i = wordcounts.begin(); i != end; i++)
      reduce(i->first, i->second);
    uploadHelper(getTaskOutputInString());
  }

  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));
  }
}
```

Figure 4: Wordcount Reduce Code

SHA1 hash. All map output files 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 with the key being the word and the value being the total number of times the word appeared in all of the ebooks processed. The final output is then uploaded back to the DataStore.

The uploads to the DataStore may include application defined parameters, such as replication that should be performed by the DataStore. It may also require that the application itself upload multiple copies its output.

4.3 Executing Nebula MapReduce Applications

Once a MapReduce application has been instantiated, the ComputePool Master 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 *.nexe 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, then the 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.

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 3.3.2.

Component	Lines of Code
Data node	1272 (Java)
DataStore Master	736 (Javascript)
ComputePool Master	750 (Python), 124 (Javascript)
Scheduler	1068 (Python)
MapReduce NaCl framework	1167 (C++)
Wordcount	193 (C++), 149 (Javascript)

Table 3: Nebula System Code Size.

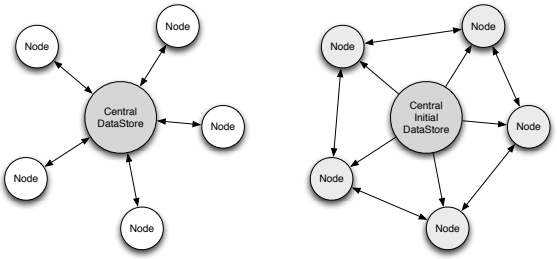
5. EVALUATION

To evaluate the performance of Nebula, and specifically MapReduce on Nebula, we have configured an experimental setup on PlanetLab [4], 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.

5.1 Experimental Setup

For our experiments, we have set up 52 nodes on PlanetLab 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. Table 3 shows the code sizes for the different components of Nebula that we have implemented.

We perform experiments using the Nebula MapReduce Wordcount application. Our input dataset consists of a set of 1500 ebooks from Project Gutenberg which amounts to



(a) BOINC (b) BOINC-MR

Figure 5: Data flow in BOINC and BOINC-MR.

500 MB of data. We expand and contract this dataset to yield different input dataset sizes for different experiments. The maximum data size 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. 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 other existing volunteer computing systems, BOINC [1] and BOINC-MR [6], on top of the Nebula infrastructure. These models have three key aspects which differentiate them from Nebula. First, in BOINC and BOINC-MR the input data is centralized. The BOINC server is usually the single source of input data, and BOINC-MR uses this same approach. 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 BOINC, intermediate data is stored centrally whereas in BOINC-MR it is stored locally on the node that performed the map task. To emulate the BOINC-MR 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 BOINC and BOINC-MR, all nodes are treated equally and tasks are assigned randomly without concern for data locality. In contrast, the default Nebula scheduler is locality-aware. The data flow for the BOINC and BOINC-MR model are illustrated in Figure 5.

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 1 for examples). Given this data placement, Nebula tries to select the best nodes for computation to minimize the overall execution time. We compare BOINC and BOINC-MR against Nebula MapReduce using both a random scheduler (Nebula-Random) and the locality-aware scheduler (Nebula-LA) presented earlier (Section 3.3.2).

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.

5.2 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 BOINC and BOINC-MR 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.

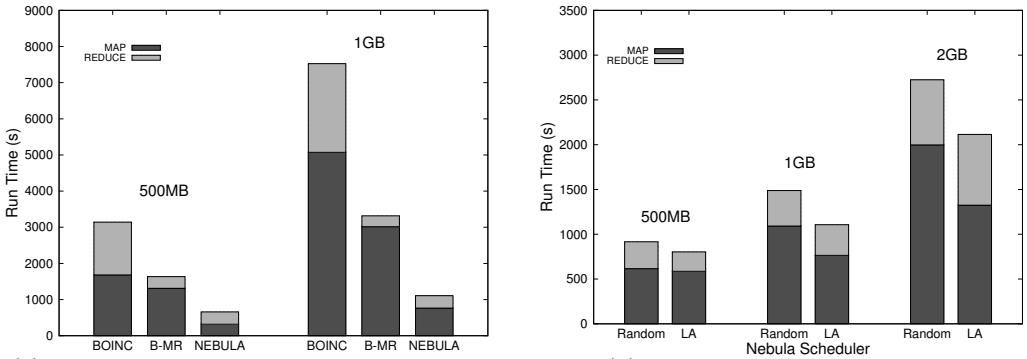
Figure 6(a) compares the Nebula model with the locality-aware scheduler against the BOINC and the BOINC-MR (B-MR in the graph) models. The results indicate that the Nebula approach is far superior to BOINC* due to the removal of data bottlenecks. For map tasks, Nebula is able to find compute nodes close to the data sources, and outperforms both BOINC and BOINC-MR which rely on a centralized data node. For reduce tasks, both BOINC-MR and Nebula exhibit good performance compared to BOINC, as they retain intermediate data locally. As the data size increases to 1GB, Nebula continues to outperform BOINC* (by 580% and 200% vs. BOINC and BOINC-MR 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 6(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.

5.3 Fault Tolerance

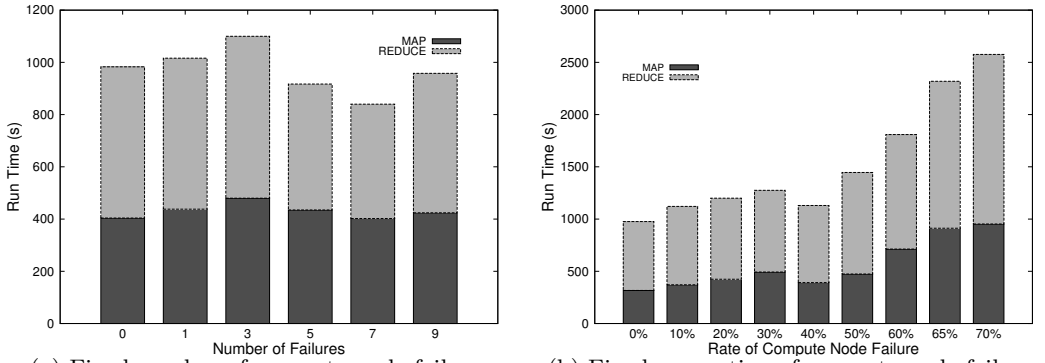
In the next set of experiments, we induce crash failures in both the compute and data nodes during execution. As discussed in Section 3, Nebula has mechanisms to provide fault tolerance in the presence of node failures. These mechanisms include re-execution of tasks to handle compute node failures and 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 6(a) and 6(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



(a) Comparison to BOINC and BOINC-MR (b) Nebula scheduler comparison

Figure 6: MapReduce comparison across different environments.



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

Figure 7: Performance of Nebula in the presence of compute node failures.

processes for these tests, rather than simulating the failures, so these experiments illustrate the robustness of Nebula in the presence of real-life failures.

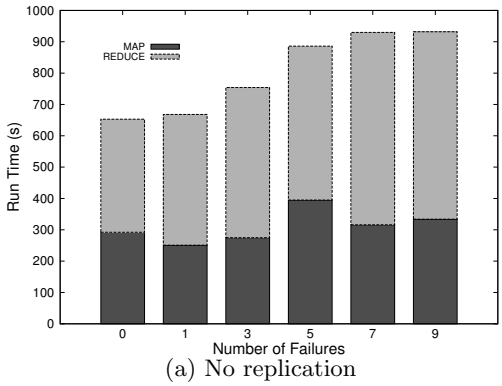
5.3.1 Compute node failures

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 7(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.

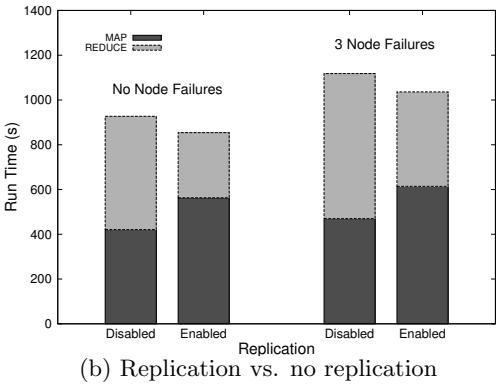
In the next experiment, we induce more severe failures, failing a large part of the system throughout the run. In this case, we randomly kill a subset of compute nodes, but allow failed nodes to come back up after a period of about 90 seconds to ensure that a fixed proportion of nodes in the system are failed at all times. Figure 7(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 are so many compute failures that the system starts becoming compute-limited, and has to re-execute many unfinished tasks.

5.3.2 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 8(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



(a) No replication



(b) Replication vs. no replication

Figure 8: Performance of Nebula in the presence of data node failures.

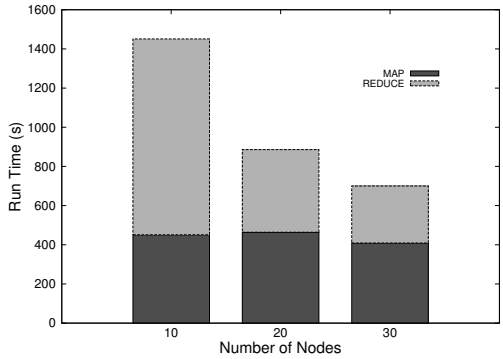


Figure 9: Scalability of Nebula.

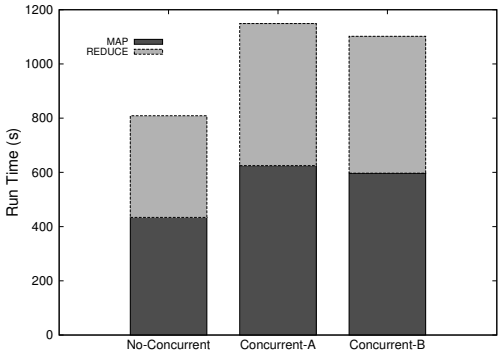


Figure 10: Concurrent Nebula applications.

with failure rate unlike the compute node failure scenarios in Figure 7(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 8(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.

5.4 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. Figure 9 shows the performance of Nebula 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.

5.5 Concurrent Applications

The final result shows that Nebula is able to run concurrent MapReduce applications. Figure 10 shows the performance of two concurrent 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.

6. 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 [5], Globus [9], 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 [11] 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 [13, 14, 10]. Moon [13] 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 [10] 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, 8, 12] would give us a more accurate view of the network as a whole.

7. CONCLUSION AND FUTURE WORK

We presented the design of Nebula, an edge-based 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 including BOINC and BOINC-MR. 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, e.g. Nebula BigTable, 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. The development of new resource partitioning techniques are planned. 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.

8. ACKNOWLEDGEMENTS

The authors would like to acknowledge grant NSF/IIS-0916425 that supported this research. They would also like to acknowledge Rohit Nair, Hon Ping Shea, Kwangsung Oh, Harold Osmundson, Jason Zerbe and Shilpi Argarwal for their efforts on building the infrastructure, earlier work on Nebula, and help with experiment design and execution.

9. 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] M. Cardosa, C. Wang, A. Nangia, A. Chandra, and J. Weissman. Exploring MapReduce efficiency with highly-distributed data. In *Proceedings of the second international workshop on MapReduce and its applications*, MapReduce '11, pages 27–34, 2011.

[3] 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.

[4] 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.

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

[6] 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.

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

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

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

[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.