Early Experience with the Distributed Nebula Cloud*

Pradeep Sundarrajan†, Abhishek Gupta, Matthew Ryden†, Rohit Nair,
Abhishek Chandra and Jon Weissman
Computer Science and Engineering
University of Minnesota
Minneapolis, MN 55455
{abgupta, rnair, chandra, jon}@cs.umn.edu, †{sunda055, ryde0028}@umn.edu

Current cloud infrastructures are important for their ease of use and performance. However, they suffer from several shortcomings. The main problem is inefficient data mobility due to the centralization of cloud resources. We believe such clouds are highly unsuited for dispersed-data-intensive applications, where the data may be spread at multiple geographical locations (e.g., distributed user blogs). Instead, we propose a new cloud model called Nebula: a dispersed, context-aware, and cost-effective cloud. We provide experimental evidence for the need for Nebulas using a distributed blog analysis application followed by the system architecture and components of our system.

1. INTRODUCTION

The emergence of cloud computing has revolutionized computing and software usage through infrastructure and service outsourcing, and its pay-per-use model. Several cloud services such as Amazon EC2, Google AppEngine, and Microsoft Azure are being used to provide a multitude of services: long-term state and data storage [27], "one-shot" burst of computation [18], and interactive end user-oriented services [17]. The cloud paradigm has the potential to free scientists and businesses from management and deployment issues for their applications and services, leading to higher productivity and more innovation in scientific, commercial, and social arenas. In the cloud computing domain, a cloud signifies a service provided to a user that hides details of the actual location of the infrastructure resources from the user. Most currently deployed clouds [12, 16, 19, 4] are built on a well-provisioned and well-managed infrastructure, such as a data center, that provides resources and services to users. The underlying infrastructure is typically owned and managed by the cloud provider (e.g., Amazon, IBM, Google, Microsoft, etc.), while the user pays a certain price for their resource use.

While the current cloud infrastructures are important for their ease of use and performance, they suffer from several shortcomings. The first problem is *inefficient data mobility:* many of the current clouds are largely centralized within a few datacenters, and are highly unsuited for dispersed-data-intensive applications, where the data may be spread at multiple geographical locations (e.g., distributed user blogs). In such cases, moving the data to a centralized cloud location before it can be processed can be prohibitively expensive, both in time and cost [3], particularly if the data is changing dynamically. Thus existing clouds have largely been limited to services with limited data movement requirements, or those that utilize data already inside the cloud (e.g. S3 public datasets). Secondly, current clouds suffer from *lack of user context:* current cloud models, due to their centralized nature, are largely separated from user context and location. Finally, current cloud models also are *expensive* for many developers and application classes. Even though many of these infrastructures charge on a pay-per-use basis, the costs can be substantial for any long-term service deployment (such examples are provided in [26, 29]). Such costs may be barriers to entry for certain classes of users, particularly those that can tolerate reduced service levels, including researchers requiring long-running large-scale computations, and small entrepreneurs wishing to test potential applications at scale.

To overcome the shortcomings of current cloud models, we propose a new cloud model [5] called *Nebula: a dispersed, context-aware, and cost-effective cloud.* We envision a Nebula to support many of the applications and services supported by today's clouds, however, it would differ in important ways in its design and implementation. First, a Nebula would have a highly dispersed set of computational and storage resources, so that it can exploit locality to data sources and knowledge about user context more easily. In particular, the Nebula will include edge machines, devices, and data sources as part of its resources. Secondly, most resources within a Nebula would be autonomous and loosely managed, reducing the need for costly administration and maintenance, thus making it highly cost-effective. Note that we do not propose Nebulas as replacements to commercial clouds - rather, we believe Nebulas to be fully complementary to commercial clouds and in some cases may represent a transition pathway.

There are several classes of applications and services that we believe would benefit greatly through deployment on such Nebulas [5]. In this paper, we focus on one such class, dispersed data-intensive services. Such services rely on large amounts of dispersed data where moving data to a centralized cloud can be prohibitively expensive and inefficient in

*The authors would like to acknowledge grant NSF/IIS-0916425 that supported this research

DIDC'11, June 8, 2011, San Jose, California, USA.

terms of the cost of network bandwidth and the time of data transfer. It would be preferable to choose the computational resources closer to the data which have sufficient computational capability.

In this paper, we describe our early experience with a Nebula prototype based on one possible implementation model: the use of volunteer dispersed resources. We first present a motivating example for the Nebula cloud model: blog analysis, a distributed data-intensive application. Our results executing this application on PlanetLab indicate the promise of our approach, in particular, how the Nebula-like[1] version of this application outperforms a centralized cloud-like version. Second, we present an inside look at our current implementation of the Nebula framework, describing some of the Nebula system components, and present some microbenchmarking results for these components. These results support the efficacy of the Nebula model and provide valuable insights into the challenges involved in instantiating it as a real-world platform.

2. MOTIVATING EXAMPLE: BLOG ANALYSIS

Blogs are a hallmark of the Web 2.0 revolution where users become content providers across the Internet. Blogs can be large containing text, audio, and video content that is dynamically changing. In some cases, blogs are aggregated at central blog servers (e.g. BlogSpot), but in general, they may be scattered across thousands or millions of Web sites. It is the latter case that motivates our work. Blogs tell us something about what people are thinking and doing and may be analyzed to identify interesting social trends. In Figure 1, we show blogs stored at three locations, and content downloaded for analysis either to a single central "cloud" or to more localized compute nodes. Note that in the distributed case, we are able to find nodes with high bandwidth to all data servers (dark arrows) due to the large availability of dispersed nodes. As shown, the path from D2 and D3 may be poor to the central cloud. A cloud version of this application may be deployed as a blog analysis service where different users can request the analysis of blogs of their choice.

Given the large number and size as well as the distributed nature of such blogs, it may be beneficial to move the analysis service closer to where the blogs are actually stored. In some types of analytics, it may be necessary to have all of the blogs in one place for tightly-coupled processing. In this case, a centralized cloud may be best. Alternatively, if the blogs can be analyzed independently (in-situ) or may require only a small degree of aggregation, for example, to determine how many blogs are concerned with politics, we need not bring them all together in one place. Additionally, if the data is changing dynamically (and thus would require extra bandwidth and cost) to update to a central cloud, it may be more beneficial to compute on it in a more distributed manner. We have developed a distributed blog analysis service in Nebula that can exploit such opportunities and is now described.

[1]We use the term "Nebula-like" to indicate that the application does not yet use all of the Nebula software machinery described in Section 4. Similarly, "cloud-like" refers to a centralized pool of resources in the same domain, and not a commercial cloud service, such as EC2. In the remainder of the paper, we drop these distinctions and refer to the models as Nebula and cloud respectively unless explicitly specified.

3. NEBULA BLOG ANALYSIS

3.1 Blog Analysis Architecture

The distributed architecture for Nebula blog analysis consists of following components: (i) master, (ii) client, (iii) data node, (iv) execution node and (v) database (see Figure 2). We now describe each component.

The master node is a central controller that acts as the interface to the blog analysis service from the user. The master node has a centralized database that has details about the execution nodes and data nodes participating in the application. The functionality of the master node is as follows: (i) it processes requests (blog analysis) from users (clients), (ii) identifies the best live execution node for a data node and then initiates blog analysis, (iii) keeps track of blog analysis progress at execution nodes, (iv) keeps track of health status of available execution nodes and data nodes, (v) sends the analysis result back to user (client) and stores it in the central DB (if required), and (vi) balances execution nodes workload. The data node(s) contain the actual blog data for analysis and are distributed geographically. Each data node is implemented as a data server that serves the blog data to an execution node upon request. The execution node serves the request from the master node to analyze the blog data for a particular data node. It downloads the blog data from the data node to perform the analysis. The best execution node for a given data node is chosen using network bandwidth information provided by a network tool described shortly. The central database acts as repository that stores the list of execution nodes, data nodes and the network bandwidth for all node pairs.

A network dashboard tool provides the network bandwidth between a data node (source) and execution node (destination). The network bandwidth found for all (data node, execution node) pairs is recorded in the central database maintained by master node. The master node uses this information to find the best execution node (having highest bandwidth) for the given data node. The network bandwidth is updated periodically in the central database using a separate service called the node bandwidth updater. This tool retrieves the network bandwidth from the dashboard for all (data node, execution node) pairs available and updates the central database. This program is run regularly to update central DB with the latest network bandwidth.

The blog analysis service accepts requests from a user to perform blog analysis by specifying the number of blogs and data node server on which blogs reside. Upon getting the request, the master node identifies the best available live execution node for each data node and initiates the analysis by sending a request to the identified execution node. The execution node then makes a connection to its assigned data node, retrieves the blog data to perform the analysis and finally returns the result to master node. The dashboard monitors nodes that are used for Nebula and exposes a Web API that enables the real time bandwidth between data node and execution to be known. We maintain a list of data nodes and execution nodes in the central database. At the start of blog analysis, the network bandwidth data already recorded in the database is used to find best execution node for given data nodes.

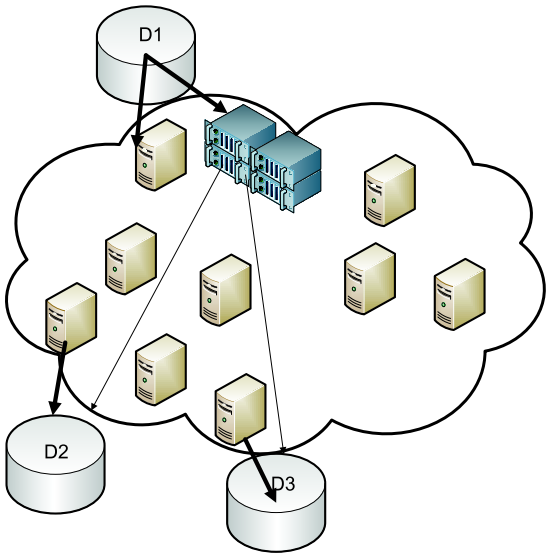


Figure 1: Distributed Blog Analysis. The nodes inside the network cloud contain individual Nebula nodes and a centralized cloud (depicted as stackable computers). The light arrows indicate low bandwidth paths and the dark, high bandwidth paths from a data source to a compute node or cloud.

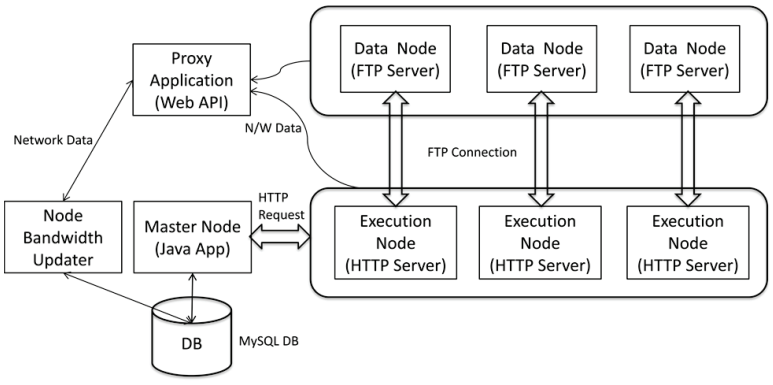


Figure 2: Architecture of Blog Analysis Service for Nebula

3.2 Experimental Results

We deployed our Nebula prototype on PlanetLab [7] as it represents a good approximation to the geographically dispersed resource pool that we envision for Nebula. We deploy various Nebula components and the blog application on it for an experimental evaluation. Within the Nebula PlanetLab slice, several of the nodes were fixed to be data nodes while the remaining nodes were considered as execution nodes. Both sets of nodes are disjoint. The system locates execution nodes that are estimated to be "close" to data sources.

The analysis code is an adaptation of Google's Blogger service written in Java [14]. We are showing results for a simple analysis: word count. The blog data was extracted from the Web using the WebHarvest Web crawler [30] to create a large blog dataset for experiments. The starting link given to the Web crawler is `http://www.blogger.com/changes10.g` that shows list of up to 2000 latest updated blogs. The blog data on all data nodes is unique.

We have compared the performance of the blog application in Nebula with a centralized cloud emulator. The centralized cloud emulator consists of nodes at same physical location in the same LAN domain. The blog analysis for all (data node, execution node) pairs was carried out in parallel during a single run. The Nebula cloud for dispersed data intensive applications was evaluated with and without failures, and for scalability.

Nebula without Failures

This experiment was conducted assuming that Nebula has no failures. The blog analysis application was run on the Nebula, centralized cloud emulator (CCE) experimental setups, and the following measurements were made: overall total time, cumulative time, blog data transfer time for each node pair, blog data processing time for each node pair, and total time to analyze the blogs for each node pair.

The overall total time is the time taken to complete blog analysis for all data nodes. The cumulative time is time taken to complete a certain percentage of analysis within a single run (useful if partial results are meaningful). The total time taken to analyze blogs for each node pair includes time taken to make an FTP connection to a data node (FTP server), browse the directory on the FTP server to get the data, retrieve the blog data (blog data transfer time), analyze the data (blog data processing time) and close the FTP connection.

PlanetLab nodes at Minnesota, California, Florida, and Europe were fixed as the four data nodes. Four PlanetLab nodes which are in same domain at Illinois were fixed as execution nodes for CCE. In the case of Nebula, for each data node, the best execution node was chosen based on network bandwidth data available in the central database. The blog analysis was conducted on both Nebula and CCE varying the count of blogs to be analyzed on each data node. The graph in Figure 3 shows the overall total time taken by Nebula and CCE to complete blog analysis for a varied count of blogs. This graph clearly shows the large difference in total time savings. The experiments show that there is a total time savings of 53% on average and significant data transfer time savings in Nebula compared to CCE. The graph in Figure 4 shows the total time savings percentage and the graph in Figure 5 shows the data transfer time savings percentage achieved for each (data node, execution node) pair in Nebula over CCE for a varied number of analyzed blogs. For the

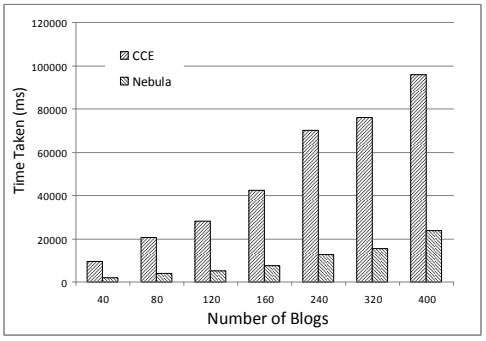


Figure 3: Overall Total Time

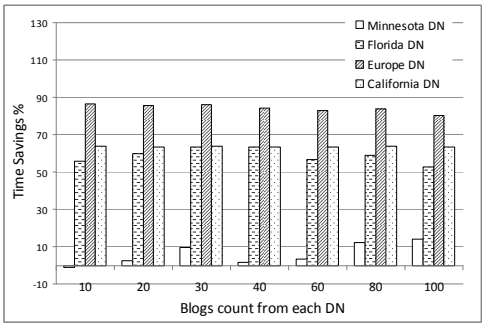


Figure 4: Total Time Savings

Minnesota DN, the selected Nebula node has a modest improvement relative to the others. For a very small number of blogs (10), it is latency-bound, and performs slightly worse than the CCE. The graph in Figure 6 shows the cumulative time taken to analyze 25%, 50%, 75% and 100% of blog data in a single run for a varied blog count. This graph shows that Nebula is ahead of CCE at all stages of blog data analysis. The reason why Nebula is far superior, is that with a single central location, there is a higher likelihood that some data node will have a poor path to the centralized cloud. In contrast, Nebula is able to select a high-quality execution node for each data node due to the network dispersion of compute nodes.

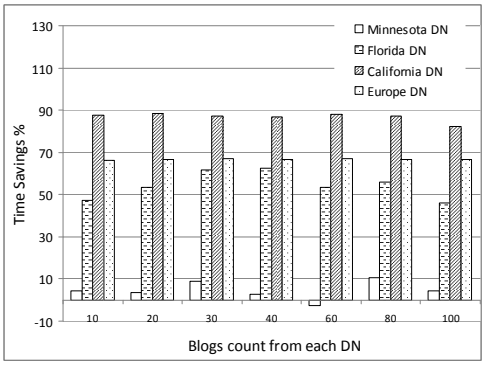


Figure 5: Data Transfer Time Savings

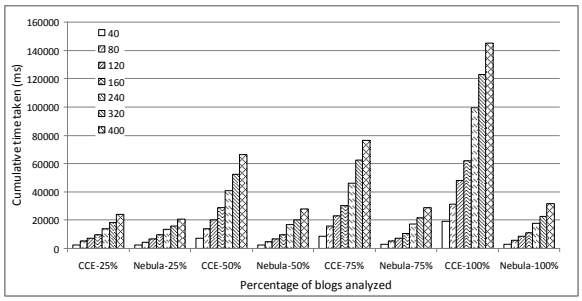


Figure 6: Cumulative Time

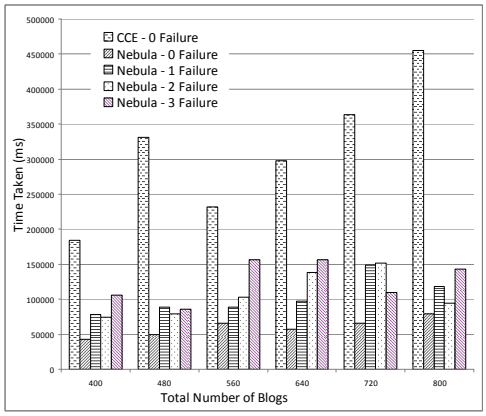


Figure 7: Overall Total Time (with failures)

Nebula with Failures

An advantage of the centralized cloud is reliability and stability. Nebula nodes based on volunteers can go down at any time and are more likely prone to failures. To see the effect of failure, we emulated failures in Nebula nodes. The failure model used in Nebula is fairly simple: failures were emulated by failing a random execution node after a random amount of time uniformly selected from a fixed range of 1-10 seconds. When an execution node fails, another execution node is selected dynamically. The experiment is run by varying number of failures for a fixed blog count. The graph in Figure 7 shows the overall total time taken to complete blog analysis for varied number of blogs. For the basic failure model, it is evident from the graph that Nebula performs better than CCE even after 3 failures.

4. INSIDE NEBULA

We now describe the Nebula system architecture and components of the software stack: distributed DataStore, network dashboard, and the secure NodeGroup. At a high level, a Nebula contains a pool of distributed resources and a control manager that we call Nebula Central (Figure 8). Nodes may be used to host data (A) or computation (B), or both. The Nebula Central is a control node (or group of nodes) that manages the participant nodes in the Nebula and the applications that are deployed inside the Nebula. It is in charge of various management tasks including resource management, code distribution, data coordination, etc. The nodes that participate in the Nebula can take one (or both) of two roles: they can act as data nodes by contributing their storage and bandwidth resources to store and cache application data, or they can act as compute nodes by contributing their CPU cycles to carry out a task execution for a Nebula application. We group together the resources provided by the participant nodes to create two abstractions: a DataStore and a secure NodeGroup, described in detail next.

4.1 DataStore

The DataStore provides the basic storage capability for application data as well as Nebula specific system/management data. It is used by applications to store their input and output data. Additionally, it may also store dynamic application-specific state that might be useful for overcoming component failures or churn. The DataStore will also serve to store and retrieve data generated by various Nebula services, such as monitoring data or events.

To support the DataStore concept as a primitive service, and also to support a wide variety of application/service needs and policies, we provide a very simple interface and storage capability partially inspired by the Amazon S3 abstraction, and similar to it, we support a flat hierarchy of data objects, where each object is identified by a unique key. In this model, we assume the objects to be immutable, i.e., we do not support deleting or editing objects explicitly. We expect a higher-level abstraction (similar to a file system) to provide such operations on top of the DataStore, e.g., through versioning and ordering/merging of concurrent operations.

For our blog analysis example, the DataStore consists of blogs which are distributed geographically. In selecting nodes to be part of a DataStore, we have to consider the relative connectivity and bandwidth among the data nodes, compute nodes, as well as external data sources from which data may be downloaded into the DataStore. Further, the DataStore needs to be reliable even in the presence of volunteer nodes. Next we describe the design of the DataStore along with the various operations supported by it. Many DataStore instances will co-exist in a specific Nebula. An application can create a DataStore with certain desired properties. In this paper, we use the term DataStore to refer to the both the concept or service and a specific instance (i.e. a subset or group of nodes storing data).

The DataStore contains nodes running the DataStore software as well as a centralized nameserver. We envision volunteers may be willing to contribute only bandwidth and storage, so the DataStore software is separate from other parts of the Nebula system. Communication between clients, nodes, and the nameserver are done using XML-RPC.

DataStore nodes are grouped for certain functionality as desired from the client. Groups can be determined using client-provided scoring functions utilizing various metrics including connectivity of the DataStore nodes and the planned computing or client nodes. These groups are then registered with the nameserver and the nodes themselves. It is important to stress that the DataStore provides a mechanism for organizing data nodes based on application-specified properties of the DataStore, such as reliability, bandwidth, etc. We do not expect the clients or even the nameserver to know which nodes are best suited for the task, but such nodes are automatically determined based on the application-desired

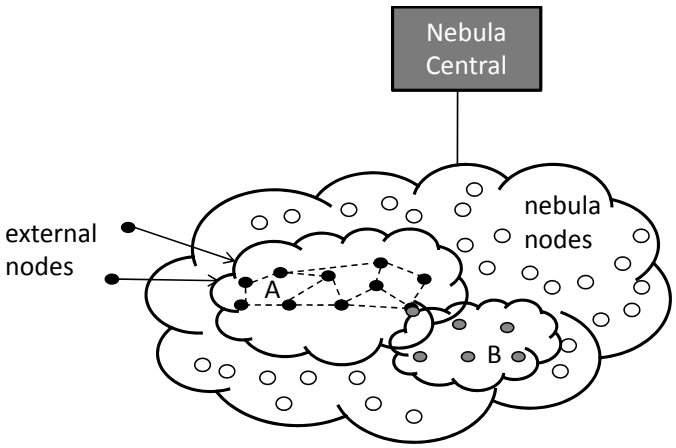


Figure 8: Nebula Architecture. A NodeGroup (B) and DataStore (A) are shown. These would be used by a particular deployed application or service. Other applications may also be deployed (not shown).

Operation	Parameters	Description
`client_create_group`	`datastore_id, source, destination, scoring_function, number`	Create data node group
`client_put`	`source, datastore_id, file_id, number`	Put a file in a data group
`client_get`	`datastore_id, file_id`	Fetch a file from a data group

Table 1: Client-side DataStore operations

metrics. We envision monitoring frameworks like OPEN [21] to provide policy implementations for the DataStore.

Table 1 shows a portion of the client-side interface to the DataStore. A client node can issue these operations to the DataStore via any Nebula node. Each of these operations has a corresponding internal operation that carries out the actual node selection, data movement, notification, etc. A typical set of DataStore operations would work as follows. First, a client interacts with a Nebula node to create a data store group for its data by using `client_create_group` operation. As part of this operation, it specifies a scoring function based on its desired metric as well as the number of nodes to be part of the group. The Nebula node receiving the client request determines which nodes best fit the scoring function by using an internal `get_nodes` operation. The selected nodes are informed of the new group using an internal `create_group` operation, and the nameserver is notified of the new group mapping as well.

With a group set up, data can now be transferred to the nodes that are part of the group. With `client_put`, the client starts a request to move a file to the group. An internal operation called `node_put` is used to download this file to a certain number of nodes in the group from the source. With `client_get`, the client requests the file from the group. Files are retrievable by clients by contacting nodes in the group directly. The client requests the nodes in the group from the nameserver.

The system has features to ensure reliability. Files are stored with a back-to-front parted hashing scheme to ensure file consistency with the ability to detect errors before the entire file has been received. The hash values are sent to the origin node on a `client_put` command to ensure consistent data is stored between nodes.

The scoring function of nodes is used to rank nodes based on a desired metric, and can be used for a variety of different use cases. For instance, if the DataStore is being used for large temporary files produced by computation nodes, a very simple scoring function may suffice:

$$\texttt{scoring\_function} = latency_{seconds} + \frac{filesize_{bytes}}{bandwidth}$$

Nodes with lower scores will have better response. This scoring policy may be suitable for a DataStore used as a temporary cache where the data is probably used once. In a second scenario, a more complex scoring function would be to needed to find a DataNode that is near both a remote data source and the client - this would be needed in the blog scenario.

As part of our prototype, a network of DataStore nodes was deployed on PlanetLab and two scenarios described above were tested. In the first scenario, the DataStore was built to act as a cache between the source nodes and the computing nodes. A file was sent from the source node to the DataStore. Later when the computing node needed the file, it was sent from the DataStore to the computing node.

The average performance results can be seen in Figure 9, which compares node selection based on the above scoring function to that for random a node selection. It took 25 seconds on average to send the file from the source to the DataStore and then from the DataStore to the computing node. When the selection was random, the average transfer time was both much higher and had a much larger variance than when it was selected for good connectivity using the scoring function. In the second scenario, a great deal of data would be transmitted between the DataStore and computing nodes (e.g. blog data). We model this as the transmission of N large data transfers, and we show N=5 in our experiments. A scoring function which weighted the transfer time between the DataStore and the computing node more heavily was used to ensure a better locality between

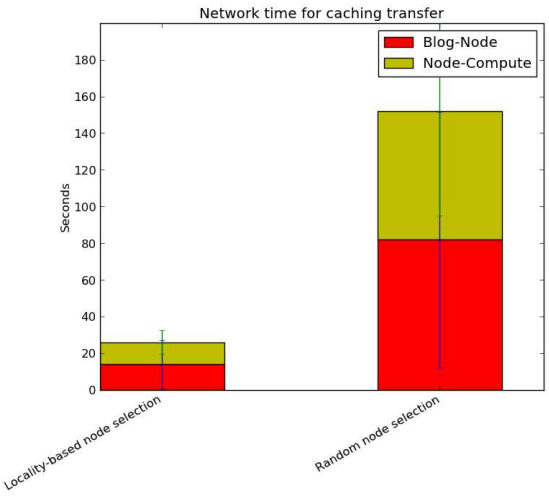


Figure 9: DataStore caching

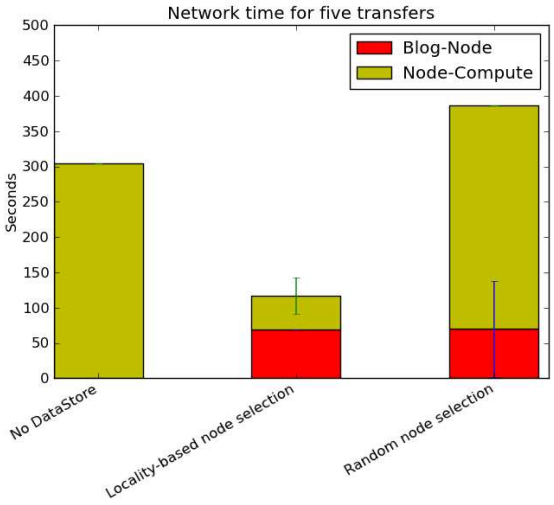


Figure 10: DataStore computation filesystem scenario

the DataStore and the computing node. The average performance results of this scenario can be seen in Figure 10. The results compare a weighted scoring policy for choosing the DataStore, a random selector for the DataStore, and a system where there was no DataStore and files were transferred directly between the source and the computing nodes. As can be seen, selecting a DataStore with a proper scoring policy was much more effective than not using a DataStore at all. Ensuring locality of the DataStore with both the source and computation node also kept the variance low in the case of a properly selected DataStore.

4.2 Secure NodeGroups

The NodeGroup abstraction identifies a related set of computational nodes to host applications and services. It is partially inspired by Amazon EC2 instances and the PlanetLab slice abstraction. The composition of a node group may change over time as nodes may be added or removed dynamically. Nodes may be allocated or deallocated to a node group, and application components may be deployed or removed from its nodes. Since Nebula nodes may host arbitrary code, it is crucial that such code be executed securely. The NodeGroup complements the DataStore abstraction.

The computation node is the node where the actual application computation happens. The computation node interacts with the Nebula Central to register itself as part of the Nebula. It contains little intelligence about the rest of the Nebula, and relies on the Nebula Central to get relevant information regarding pending tasks and DataStore nodes that are close to it.

The computation nodes, just as the DataStore nodes, consist of voluntary resources that can join and leave the Nebula on an ad-hoc basis. In our initial prototype, we have focused on three issues: a) security b) performance and c) cross platform support. We propose a system by which any user with an Internet connection and a modern browser like Chrome can volunteer as a compute node. To join the Nebula as a compute node, the user can just open a Web page on his browser and start the computation. To stop the computation, the user just needs to close the page or the browser.

The main component that we use to implement a secure compute node is *NativeClient*, a browser plugin developed by Google [15] to achieve portability and sandboxing for running native code in Web applications. An appealing feature of this approach is that a participating machine only needs only to run a browser to securely participate vs. a more heavy-weight solution such as virtual machines. We have integrated NativeClient into our system as part of our Nebula implementation. Nebula applications are written in C/C++ and compiled against the NativeClient specific compilers provided by the SDK. On compiling, we get NativeClient compatible binaries, or .nexe files. These .nexe files can now be embedded in Web pages, and any browser with the plugin enabled can execute them. In other words, the compute node is nothing but a regular computer where the user has opened the Web page hosted by the Nebula Central in his browser.

Depending on the application, the browser downloads the appropriate .nexe file from the server. Once the user directs the page to start the computation, it retrieves pending tasks and details about the data from the server. On receiving this information, it begins pulling data from the DataStore nodes asynchronously and passes it on to the actual native

code. We have currently implemented this in JavaScript, although in the future, this can be handled by the application code itself. The application code then executes on the client machine and returns the results back to the server. As described earlier, it is the job of the DataStore to assign nodes that are most optimal for the given compute nodes, such that there is minimal latency in data transfer.

All of the interactions between the components are standard HTTP calls. The data exchange format used is JSON, which has the important characteristic of being lightweight in comparison to other formats like XML. We define a few basic REST-based Web services for this implementation. The first one retrieves pending tasks for a particular application. This service also returns a list of URLs from where data can be retrieved as needed by the task. The JavaScript then makes regular AJAX calls to retrieve the data and passes it on to the NativeClient plugin. The application code then does the required computation, and returns the result back to the JavaScript layer. The JavaScript then makes another Web service call to report back the result.

This implementation implies that the user does not have to explicitly download potentially untrusted software to contribute to the computation. The application code is downloaded by the browser and it is sandboxed within the NativeClient architecture throughout it's entire execution cycle. The sandbox implements some constraints on the NativeClient binaries by which it can ensure security. NativeClient applications have performance comparable to that of the equivalent independent native code binaries. The primary overhead is the context switch from the browser to the application code. There are examples where similar compute nodes have been implemented, but with the computation done by JavaScript to achieve sandboxing [9]. Although such implementations might be simpler, the performance would start degrading once the nature of the computation becomes more complex and resource heavy.

Figure 11 shows the overhead of using NativeClient with a C benchmark code relative to executing the native code directly. We used two types of benchmarks in this experiment. The first experiment uses a simple program to calculate the number of steps it takes to obtain the Collatz Sequence [10] of a number. The program repeatedly calculates this for numbers from 1 to a limit defined by the program. This uses a simple recursive algorithm and performs some basic mathematical operations. The nature of the algorithm is also such that it is guaranteed to terminate. For our comparison, we've started with a limit of 1 million numbers, to a limit of 5 million numbers. The code is then run outside the NativeClient container, as a regular executable, and then within the container via Chrome. The execution times for these form the basis of our comparison. It is worth noting that executing the same code with JavaScript takes over 10 seconds for 1 million numbers. We can thus safely ignore that implementation from our comparison.

The second experiment is closer in nature to blog analysis and performs a variety of text based computations[2]. We added some changes to the simple word count application to show that code running withing NativeClient can indeed scale with increasing complexity. The program takes in text as input and then breaks it down to individual words

[2]We are the process of porting the Java-based Google blog analysis to native code for an end-to-end integration with NativeClient and the rest of the Nebula software stack.

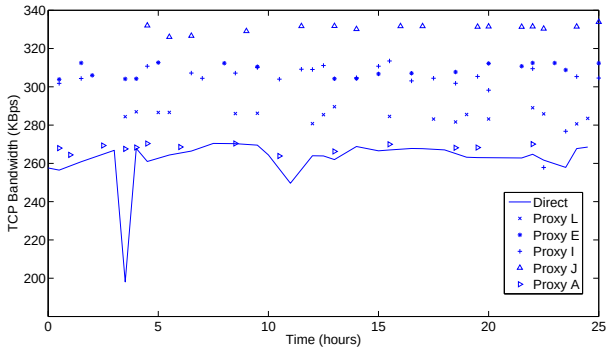


Figure 12: Dashboard TCP Statistics

using the standard C `strtok` function. The next step is to insert each word into a linked list in alphabetical order. This involves calls to the `calloc` and `strcmp` functions. The list is then traversed to find the number of words, which is equivalent to the number of nodes. Finally, the dynamically allocated memory is released using the `free` function. The added functionality is not required for calculating the word count, but demonstrates frequent memory accesses and other functions that you might use in a typical blog analysis application. For our benchmarking, we provide inputs of different word counts and measure the execution time.

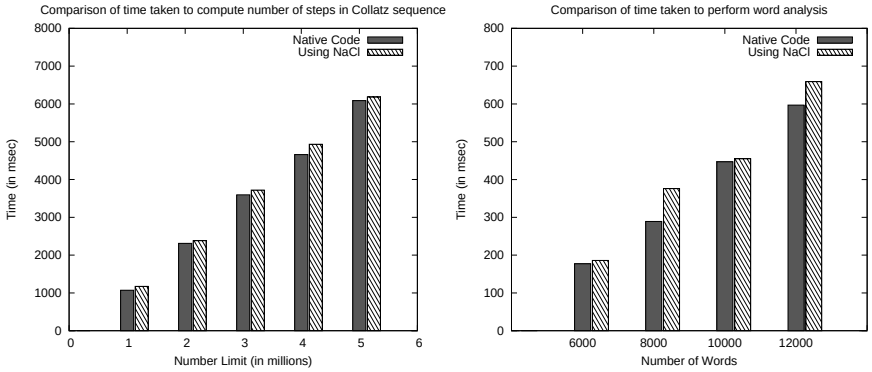
As can be seen from the figures, the overhead ranges from 2% to 30% (worst case). We believe this overhead is acceptable given the isolation and security provided. Another point to note is that the overhead does not seem to increase based on increasing input size.

4.3 DashBoard

The network dashboard (`netstat.cs.umn.edu`) is a set of network monitoring tools deployed across all participating Nebula nodes. The dashboard periodically measures TCP, UDP, bandwidth between node pairs, as well as jitter and mean delay. The dashboard can also identify overlay network paths that can out-perform the default Internet routing paths [25]. The data collection methodology is simple. Every few minutes, our program wakes up, queries our central server to locate the addresses of other nodes running the same program. This stub then takes point-to-point measurements and reports it back to the central server. The data is stored in a network database. When a node joins the Nebula, it downloads the necessary dashboard components which run inside NativeClient. A sample of the dashboard data is shown in Figure 12 for two endpoints using TCP (additional paths using routing proxies are also shown).

The dashboard can be used by Nebula to select nodes for deployment, identify network bottlenecks, replace failed nodes, depending on the needs of the Nebula application. For example, some applications may require low latency or high-bandwidth between nodes if they interact frequently. In other cases, e.g. a replicated stateless service, little interaction may be needed, thus, the network requirements may be minimal.

5. RELATED WORK



(a) Collatz (b) Text analysis

Figure 11: Performance overhead of NativeClient

Our projects is related to a number of different works. Volunteer edge computing and data sharing systems are best exemplified by Grid and peer-to-peer systems including, Kazaa [28], Bittorrent [8], Globus [13], BOINC [2], and @home projects [23]. 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 typically used to host persistent services that one may envision in the cloud.

Estimating network paths and forecasting future network conditions are addressed by [31]. We have used simple active probing techniques and network heuristics for prototyping and evaluation of network paths in our dashboard. Existing tools [24], [11], [22] would give us a more accurate view of the network as a whole. Direct probing in a large network isn't scalable, and we advocate the use of passive or secondhand measurements [21]. [20] shows that it is possible to infer network conditions based on CDN [1] redirections and [6] is an implementation of such a scheme.

6. CONCLUSION

In this paper, we have presented Nebulas, a new cloud architecture designed to support distributed data-intensive applications using volunteer nodes. We described the Nebula system architecture and software services needed to support Nebula applications: network dashboard, DataStore, and secure NodeGroups. We used a distributed blog analysis application to illustrate the potential performance benefits of the Nebula cloud vs. a centralized cloud. The result show that a Nebula can be beneficial even in the face of node failures. Future work is targeted at completing the Nebula software stack and to begin to deploy real applications.

7. REFERENCES

[1] Akamai. `http://www.akamai.com`.

[2] 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.

[3] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the clouds: A berkeley view of cloud computing, Feb 2009.

[4] Azure services platform. `http://www.microsoft.com/azure/default.mspx`.

[5] A. Chandra and J. Weissman. Nebulas: Using distributed voluntary resources to build clouds. In *HotCloud'09: Workshop on Hot topics in cloud computing*, Berkeley, CA, USA, June 2009. USENIX Association.

[6] D. Choffnes and F. Bustamante. On the effectiveness of measurement reuse for performance-based detouring. In *INFOCOM 2009, IEEE*, pages 693 –701, april 2009.

[7] 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.

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

[9] Collaborative map reduce. `http://code.google.com/p/nativeclient`.

[10] Collatz Conjecture. `http://en.wikipedia.org/wiki/Collatz_conjecture`.

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

[12] Ec2. `http://aws.amazon.com/ec2`.

[13] 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.

[14] Google blogger service. `https://www.blogger.com/start`.

[15] Google native client. `http://code.google.com/p/nativeclient`.

[16] Google app engine. `http://www.google.com/appengine`.

[17] Google maps. `http://maps.google.com`.

[18] D. Gottfrid. Self-service, Prorated Super Computing

Fun! http://open.blogs.nytimes.com/2007/11/01/self-service-prorated-super-computing-fun.

[19] Ibm cloud computing. http://www.ibm.com/ibm/cloud.

[20] A. jan Su, D. R. Choffnes, A. Kuzmanovic, and F. E. Bustamante. Drafting behind akamai (travelocity-based detouring. In *In Proceedings of ACM SIGCOMM*, pages 435–446, 2006.

[21] J. Kim, A. Chandra, and J. Weissman. OPEN: Passive Network Performance Estimation for Data-intensive Applications. Technical Report 08-041, Dept. of CSE, Univ. of Minnesota, 2008.

[22] 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.

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

[24] A. Pasztor and D. Veitch. Active probing using packet quartets. In *In ACM SIGCOMM Internet Measurement Workshop*, pages 293–305, 2002.

[25] S. Ramakrishnan, R. Reutiman, A. Chandra, and J. Weissman. Standing on the shoulders of others: Using proxies to opportunistically boost distributed applications. Technical Report 10-012, Dept. of CSE, Univ. of Minnesota, 2010.

[26] G. Reese. *Cloud Application Architectures: Building Applications and Infrastructure in the Cloud* . O'Reilly Media, Apr. 2009.

[27] Public Data Sets on AWS. http://aws.amazon.com/publicdatasets.

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

[29] P. B. Teregowda, B. Urgaonkar, and C. L. Giles. CiteSeerx: A Cloud Perspective. In *HotCloud'10: Workshop on Hot topics in cloud computing*, June 2010.

[30] Web-harvest web crawler. http://web-harvest.sourceforge.net.

[31] 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*, 15:757–768, 1999.