Ensuring Reliability in Geo-Distributed Edge Cloud

Albert Jonathan, Muhammed Uluyol, Abhishek Chandra, and Jon Weissman
Department of Computer Science and Engineering
University of Minnesota - Twin Cities
Minneapolis, Minnesota, USA
{albert, uluyo002, chandra, jon}@cs.umn.edu

***Abstract*—Centralized cloud platforms have been widely utilized for data-intensive computing in many domains. However, such systems are not suitable for geo-distributed applications since they require data to be moved to a central location for processing. Recent works have proposed an alternative cloud platform, called *edge cloud* that provides computational and/or storage resources at the edge, enabling in-situ data processing and low latency. Although such a dispersed cloud model offers low latency, it comes with reliability trade-offs. First, edge resources are interconnected using a wide-area network which is less reliable compared to an intra-cluster network. Second, resources in the edge cloud are typically highly heterogeneous leading to performance variability. Third, edge resources may span different organizational domains, containing different participation rules, leading to greater unreliability. In this paper, we discuss the issues of reliable computation and data storage availability in a geo-distributed edge cloud system built using commodity resources. We introduce a notion of *reliability factor* which defines how reliable a node is. Using this reliability factor, we schedule tasks to a set of nodes to meet a certain reliability goal and dynamically replicate data to achieve timeliness for computation and high data availability for data storage respectively. We evaluate our techniques on the Nebula edge cloud and find that the use of reliability factor results in better performance and storage utilization.**

I. Introduction

Centralized cloud systems [1], [2] are the de-facto platform for executing large-scale data analysis in many domains. The convenience of cloud systems is attractive - a user can simply request resources (e.g. storage space or machines) as needed. No physical hardware needs to be managed, and typically minimal software setup is required. Yet, due to the centralized nature of cloud systems, data must be brought into a central location for processing which is inappropriate for many data analytic applications whose data itself is produced in a distributed fashion around the world. For example, content distribution networks (CDN) and large web services produce logs in a large number of sites distributed over the globe which must then be processed for anomaly detection, billing users, or for other kinds of analysis.

Most large-scale data analytic applications produce output results that have substantially reduced volume compared to the size of the input data, typically through aggregation or filtering. For example, most video feeds contain uninteresting data and only anomalies need to be analyzed. For this reason, processing data closer to the input source can reduce data transfer overhead, and therefore increase overall system performance.

Edge cloud [3]–[7] offers a wide distribution of storage/-compute resources around geographic locations. It can provide in-situ data processing for geo-distributed data analysis and low latency to the data sources. However, such systems face additional challenges: 1) the links between computational resources are typically over a wide-area network (WAN), 2) the types of machines involved may vary greatly from one another since they can be provided and maintained by different administrators unlike the centralized cloud's machines, and 3) edge failures occur with high likelihood [8], [9]. Thus, there is a need to guarantee a certain level of reliability for such systems. The use of an edge cloud for data storage can provide low latency, however, ensuring that files are stored with high availability becomes more challenging due to the reasons mentioned above. Further, determining how many replicas should be maintained for each file to ensure high availability needs to be carefully considered since storage resources are typically limited. Thus, the system needs to not only ensure that data are stored with high availability but it also must establish a replication policy to efficiently manage and utilize the storage resources.

We address the reliability issues for computation and data storage on a heterogeneous, dispersed edge cloud system called Nebula [7]. We introduce a notion of *reliability factor* that defines the reliability of a node. The reliability factor applies to both a compute and data node. In the case of a compute node, we consider a node to be be reliable if it can return correct results in a timely manner. To ensure a high reliability of processing a task, we incorporate redundancy in task scheduling to a group of compute nodes that meet the *reliability goal*. In the case of a data node, the reliability factor defines the probability the node being online or available. This data node reliability factor is used to ensure data availability in the system. We implement a file replication technique that dynamically determines how many replicas for each file should remain online at anytime and where each of them should be stored based on data nodes' reliability factor. We show that our techniques can better meet the availability goal with lower storage overhead and result in a better performance compared to the common static replication technique.

II. Background: Nebula Edge Cloud

We provide a background of Nebula [7]: the edge cloud that we consider throughout the paper. Figure 1 shows the Nebula system architecture. In this paper, we focus on the DataStore

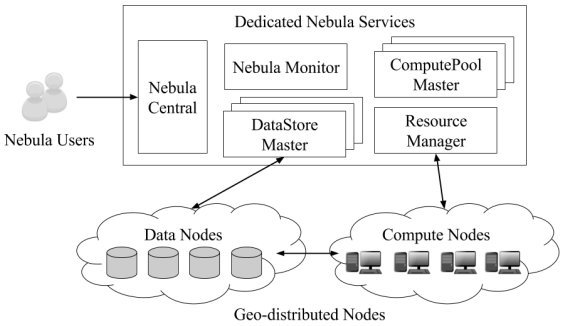


Fig. 1: Nebula system architecture [7]

and ComputePool whose respective masters are responsible for ensuring data storage availability and task execution reliability respectively. Other components are explained in [7].

• **ComputePool**: ComputePool provides per-application computational resources through a set of compute nodes. Compute nodes within a ComputePool are scheduled by a *ComputePool Master* that coordinates their execution. When scheduling tasks, the ComputePool Master will attempt to maximize data locality and avoid selecting low-performance nodes which can be measured based on their reliability factors. The compute nodes use the DataStore to access and retrieve data.

•**DataStore**: DataStore is a per-application storage service that supports efficient and location-aware data storage in Nebula. Each DataStore consists of data nodes that store actual data, and a *DataStore Master* that keeps track of the storage system metadata and makes data placement decisions.

Throughout the paper, we will only focus on MapReduce [10] applications. However, our techniques are not limited to Nebula nor the MapReduce computing framework.

III. Compute Reliability

Task processing in a system built from unreliable resources provides no guarantee whether the computation scheduled on a compute node will complete the execution nor that it will do so in a timely manner. Compute nodes that have substantially lower performance relative to their peers are called stragglers and are a well-known issue that hinder performance in many distributed systems [11], [12]. Stragglers may occur for any number of reasons, including misbehaving hardware, slow hardware, and abnormally high system load. Stragglers are examples of performance failures which have been shown to deteriorate system performance substantially - typically more than complete failures would.

We solve the issue of compute reliability in Nebula by combining the notions of *timeliness* and *correctness* which results in a decrease in overall time-to-completion while increasing confidence in the results. We established a notion of a *reliability score* (will be discussed in III-A) that is maintained per-node which defines how reliable a node is. Every node's reliability score is considered by the ComputePool Master scheduler during task scheduling. Furthermore, the scheduler may also decide to schedule each individual task to groups of nodes pro-actively rather than individual ones. This redundant task deployment is useful to mitigate the performance degradation as a result of scheduling the task to a straggler.

Pro-active approaches often sacrifice average-case performance for better worst-case performance. If this trade-off comes with low enough overhead, then it becomes advantageous to take the pro-active approach. When it comes to detecting stragglers, a passive system needs to observe a performance failure before it can attempt to mitigate the issue. Due to the inherent variability in many distributed systems, such approaches will inevitably be either inadequate or overly complex. The problem of solving the straggler then effectively becomes the problem of minimizing the overhead needed to actively avoid performance failures.

A. Compute Node Reliability Score

The reliability score of a compute node is an estimate used by the ComputePool Master that predicts the probability that the node will return a result in a timely manner. We focus on timing errors only, byzantine errors are outside the scope of this paper. The ComputePool Master computes approximate deadlines for tasks when they are assigned to nodes. This is used to estimate the running time of future tasks and to reduce the reliability score of nodes that cannot complete their tasks by the deadline. We allow 30% leeway which permits tasks to run longer than the original estimate since most of the tasks can be completed within the extended time period.

The ComputePool Master maintains counters of the total number of tasks scheduled on a compute node and the number of responses generated within the approximate deadlines. The reliability score can then be computed as the percentage of tasks that are completed within the deadlines. To avoid unrealistically high or low reliability scores, dynamic limits are applied to the reliability scores. Let r_i represent the reliability score of compute node i, b_{max} represent the maximum deviation that we are willing to tolerate, $\bar{r}$ be the sample mean, and s represent the corrected sample standard deviation. If $r_i < \bar{r} - b_{max}s$, then compute node i will be ignored during scheduling due to its lack of reliability. If $r_i > \bar{r} + b_{max}s$, then we set $s_i = \bar{r} + b_{max}s$ to avoid using reliability scores that are unrealistically high. Very high scores are likely an artifact of our scoring mechanism and not a reflection of reality.

B. Task Scheduling

The ComputePool Master scheduler implements locality task scheduling which will schedule tasks closely to the input data location. The closeness here is a measurement of network bandwidth that is monitored and periodically updated by a network monitoring service. To ensure that a task meets a certain reliability goal, r_{goal}, the scheduler incorporates some redundancy by deploying each task to a group of nodes rather than to a single node until the following equation holds:

$$1 - \prod(1 - \hat{r}_i) \geq r_{goal} \quad (1)$$

Intuitively, by scheduling a task to multiple nodes, the probability of multiple nodes going down at the same time is

significantly lower than that of a single node. It also improves the likelihood that the task finishes within the time bound.

We use a Random-Fit algorithm for selecting additional nodes. Here, the reliability goal can be set as a per-application parameter based on how critical the application is. However, this needs to be set carefully. Setting a reliability score too high in an unreliable environment may degrade the overall job completion time since the task scheduling may result in the creation of too many redundant tasks which will limit the number of distinct tasks running in parallel.

IV. Data Availability

A common technique to ensure data availability is to maintain a number of replicas for each file. For example, HDFS [13] uses a *replication factor* of 3 for each file by default. In this case, the DataStore Master would periodically check that for every file, 3 replicas remain online. If any of the replicas is lost due to failure in a data node storing the replicas, the DataStore Master would create additional replicas. This is a straightforward policy to implement, but using a fixed number of replicas for every file is not an appropriate solution in a very dynamic and heterogeneous environment.

To ensure that files remain available with high probability, the system requires knowledge of the reliability of each data node to determine the replication factor accordingly. The reliability of a data node can be estimated based on the probability of the node being online. In fact, it is desirable for a client of the DataStore to provide an *availability goal* and let the system ensure that goal is met. The availability goal defines the probability that the files remain online.

Alternatively, the availability goal can be implemented on a per-file basis rather than on a global basis. Having per-file availability goals has similar complexity to using a global availability goal but has an additional advantage: It provides the user with a parameter to adjust the trade-off between high availability and low replication and storage overhead on a per-file basis. This has many practical applications. For instance, in a MapReduce application, one might set the output of the map tasks to have a relatively low availability goal compared to the reduce tasks since that data is short-lived. The input data to a MapReduce job would likely have an even higher goal than the output of reduce tasks since the MapReduce job could always be re-executed to retrieve the lost data.

Many data storage systems take advantage of erasure codes to reduce storage overhead [14], [15]. Applying erasure coding increases the size of data in exchange for redundancy. It allows for some k, where k is a configurable parameter, chunks to be lost while maintaining the ability to recover the original data with high probability [16]. While convenient and more storage efficient than making full copies of the data, this is impractical for many use cases. When erasure codes are used, a sufficient number of chunks must be downloaded to reconstruct the original file, the erasure coding must be applied again, and the missing chunks need to be placed on new data nodes. This process consumes a substantial amount of network bandwidth, CPU time, and memory. In systems with relatively low failures, using erasure coding can have substantial benefits [17] but for the purposes of Nebula, we cannot assume this to be the case. For this reason, we opt to make full copies of the files stored in the Nebula DataStore.

A. Estimating Availability

The data nodes send heartbeats to the DataStore Master to indicate that they are online. When a heartbeat has not been received by a predetermined deadline, the DataStore Master marks the node as being offline. A node i's history of being online is used to compute its observed probability of being offline $\hat{P}_{down,i}$. This per-node score has a low accuracy when a data node first joins the system. In particular, it is likely to be an overestimate of the true availability of the node. So, we cap its value at 0.95. If $\hat{P}_{down,i}$ is ever an underestimate, this results in worse utilization but still meets the requested availability goal, so we do not adjust for this. In order to choose a set S of nodes to replicate the file onto, new nodes are added until the following inequality holds:

$$1 - \prod \hat{P}_{down,i} \geq P_{available} \tag{2}$$

The intuition here is the same as the use of reliability scores that is used by ComputePool Master to add redundancy in task scheduling (Equation 1). Using this equation gives a benefit compared to having a static replication factor. In the case of data nodes having very low availability score, this technique will give a higher file availability compared to the static replication. On the other hand, if most of the nodes have high availability scores, this will result in saving of storage resources since a smaller number of replicas would be created.

Alternatively, we may use a *global availability score* $\hat{P}_{down}$ for all nodes when determining the number of replicas that needs to be maintained. The global score can be computed as the percentage of online nodes. This is a simplification, but in environments where adequate data cannot be found on most nodes (and default values are used as placeholders), such a policy can be superior to one using per-node scores. As in the case of the per-node scores, we cap the value of $\hat{P}_{down}$ to avoid artificially high availability scores. With this global availability score, we can compute the number of nodes n necessary to meet the availability goal by solving the following equation:

$$n = \lceil \frac{log(1 - P_{available})}{log\hat{P}_{down}} \rceil \tag{3}$$

Choosing which n nodes to use can be implemtyented using a different node selection policy. We have implemented different data node selection policies for data upload [7] and similar policies can be used for selecting data nodes for the replicas.

The data node selection methods described above make prioritization of data nodes an independent problem because they are independent of what order the nodes are provided in. This makes it possible to prioritize nodes by ordering them in decreasing order. The prioritization that we use is based on bandwidth estimates between the client requesting the upload and the data nodes. For data nodes for which no bandwidth estimates exist, we randomly insert them into the list.

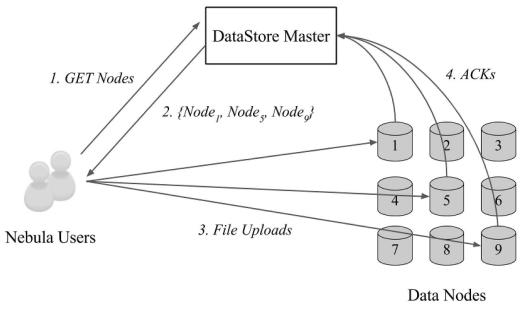


Fig. 2: File upload with redundancy

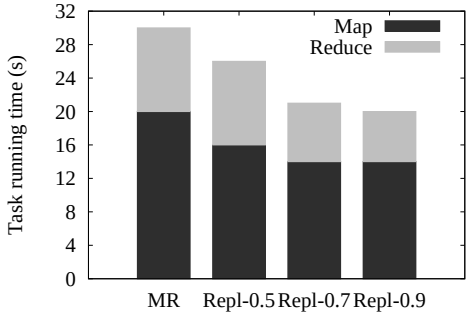


Fig. 3: Average task running time over replication factor

The actual selection of nodes takes place on the DataStore Master as shown in Figure 2. This is done for a number of reasons: it consumes less bandwidth since only the nodes that should be used for upload to are sent to the client, it simplifies the centralized work that the DataStore Master needs to do, and it provides a clean mechanism to refresh stale data. After the client receives the list of nodes, they upload the file to the data nodes which in turn informs the DataStore Master that they have received the file. The DataStore Master ensures that a sufficient number of replicas are available.

V. Experiment

We deployed 16 compute and data nodes on PlanetLab [18]. All the Nebula service components were all run on a machine with an Intel Xeon CPU E5-2609 and 16 GB of memory.

A. Compute Node Reliability

In this experiment, we used a MapReduce wordcount program as our application with a 128 MB input file. The file was split into 10 MB chunks that were randomly distributed across the data nodes, and we created one map task per chunk. For our modified scheduler, we enforced a minimum of two compute nodes per reliability group and a maximum of six. Additionally, we set $b_{max} = 2$ to enforce that reliability scores be no more than two standard deviations away from the mean. We then compared the original scheduler to ours using reliability goals of 0.5, 0.7, and 0.9. We refer to the scheduler that does not consider reliability as 'MR' and our modified versions as 'Repl'.

Figure 3 shows that the use of a higher reliability goal results in a decrease of the average task running time. This suggests that the modified scheduler is less susceptible to variation in performance, and that in the face of stragglers, the modified scheduler was able to maintain its performance characteristics better than the original Nebula scheduler. The performance gain did not result from the scheduling policy itself, as all the tasks in all cases are scheduled using the same locality scheduling.

B. Improving Data Availability

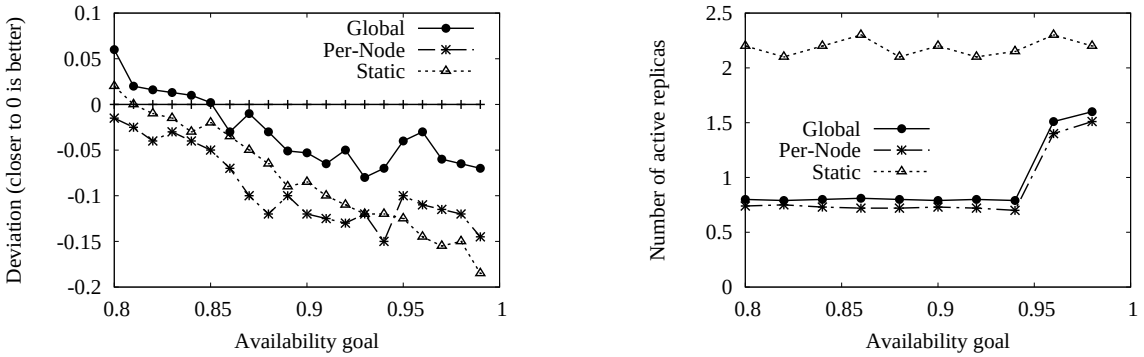
In this experiment, we compare the use of a global availability score and per-node scores for meeting user-supplied data availability goals. In both cases, the DataStore Master ensures that a sufficient number of replicas exist online. Even if all of the data nodes holding replicas for the files remain online, more replicas may be created if the availability score drops below a certain threshold. This is useful especially in the case of the global score as a node going offline may decrease the score, requiring that new replicas be made. We also compare against a naive policy with a replication factor of 3 when the file is stored into the DataStore without dynamically creating more if any replicas go offline. We refer to this policy as static replication. We deployed Nebula on PlanetLab and Google Compute Engine [19] with 67 data nodes on PlanetLab and 8 compute nodes on Google Compute Engine.

1) Meeting Availability Goals: To introduce failures in our experiments, we created a daemon that causes random failures. After a random wait of at most 2.5 minutes, the program would make a selection of nodes to kill following a power-law distribution. Of these, 90% of them were randomly selected to be restarted after waiting for a maximum of 2.5 minutes. Before beginning any of the experiments, we introduced a delay to allow the DataStore Master to adjust availability estimates with the random crashes being run in the background. After the delay, 2,000 files were uploaded into the DataStore with availability goals ranging between [0.8,1).

Figure 4(a) compares the mean achieved availability against the desired availability goal. A file was considered available if at least one replica remained online. As shown in the figure, the global availability score resulted in the highest achieved availability. Interestingly, using the per-node scores was worse than the static replication except for nodes requesting better than 95% availability. This is likely due to the inaccuracy in early measurements of reliability score for each node which may converge in time. The change at 95% is likely due to the cap placed on the availability scores as the achieved availability using the global score also had a jump at this point. Having the cap at 95% forces the creation of multiple replicas to achieve high availability.

Figure 4(b) shows that the per-node and global scores resulted in far lower number of replicas. In the case of the per-node scores, a new replica will only be created if one of the nodes storing the replicas is lost. We observed that the overwhelming majority of online nodes have a high availability score suggesting that only a small portion of files that are stored on unreliable data nodes will be replicated.

With global scoring, replicas will be distributed more evenly across the data nodes whereas the per-file scoring will favor



(a) Mean difference between achieved and desired availability

(b) Average number of replicas per file

Fig. 4: Comparison of different replication policies to meet availability goals

to store replicas in a smaller number of nodes with high availability. This results in a smaller number of replicas created in the latter case since the files that are stored in a highly available data nodes may not even be replicated if their scores have met the availability goal. However, relying on a smaller number of nodes with high availability results in larger average deviation compared to distributing the files more evenly (as shown in Figure 4(a)) since failures on any high availability nodes will result in larger number of files becoming unavailable.

2) Performance Impact: We measured the performance impact of the three replication policies using the Nebula MapReduce wordcount application with 1 map task per input chunk and 8 reduce tasks. We used an availability goal of 0.999 for the input data, 0.95 for the intermediate results, and 0.999 for the reduce tasks. We used a lower goal for the map tasks because the output of the tasks is short-lived and does not need to be available for a very long time. Our input data was constructed by randomly selecting connectives, proper names, words, and two-word phrases from the public domain Websters Second International Dictionary. The copy used is maintained by FreeBSD and was checked out in June, 2010. We used input sizes of 64MB and 128MB and split the data into 1 MB chunks. The reason we use datasets of this size is because of PlanetLabs bandwidth limits which is unlikely to be an issue in a real system. For this test, we did not induce random failures as we are analyzing performance, not availability of data.

Figure 5 shows that the dynamic replication results in a lower running time. As all replication policies use the same prioritization schemes for data nodes, the benefit is unlikely due to enhanced locality of data. Instead, this is due to the reduced number of replicas compared to static replication since uploading multiple copies of the data takes additional time.

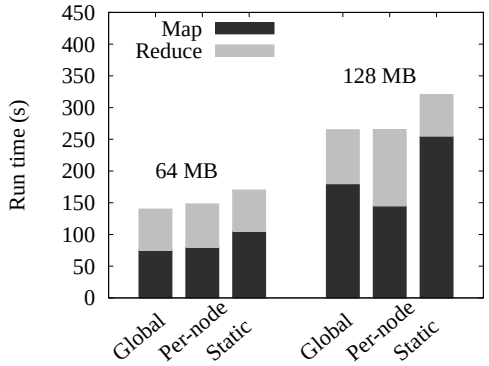


Fig. 5: Different replication policies impact to running time

VI. Related Work

There have been a number of efforts that have considered the use of the edge, but most of them focus on achieving better performance by optimizing task scheduling by minimizing network overhead without considering the nodes itself [20]–[23]. There are also projects that have looked at the use of geo-distributed storage system [24]–[27]. However, they mostly focus on an inter-data center environment and focus on optimizing latency access to the end users or minimizing monetary cost and simply rely on the underlying distributed file system for reliability.

There have been projects that have looked at the problem of building a reliable distributed system using unreliable nodes. BOINC [28] is popular volunteer-based system for crowdsourcing computational resources to do scientific research. A system called RIDGE [9] was developed for BOINC that combined the notions of correctness and timeliness to redundantly schedule tasks on multiple volunteers and cross-verify the results to the benefit of both metrics. Google File System [29] and Amazons Dynamo [30] have looked at this issue in the context of datacenter computing. These systems have the property that failures within a cluster are benign and that machines will remain available for longer periods of time and with lower variance than in edge cloud systems.

There are many projects that have looked at the issue of providing reliable data storage in non-datacenter environments including peer-to-peer systems, volunteer-based systems, and utility-like data storage systems [14], [15], [31]. The challenges faced by these projects vary substantially. In OceanStore, only the user is trusted with the non-encrypted data, and the system has to provide mechanisms to encrypt and replicate the data in a manner that keeps the contents of the data available while protecting the data and encryption keys from being exposed to any third parties including those

responsible for storing the data. Many systems place high levels of fault-tolerance above all other concerns. They work to ensure that data remains available even in the face of Byzantine failures [32]. In addition to providing Byzantine fault-tolerance, Glacier monitors the number of machines that fail and increases the amount of replication to tolerate the high failure rates. Although much of this work is relevant to Nebula, we find it desirable to have the user provide availability goals based on the needs of the data that is being stored.

VII. Conclusion and Future Work

We explored the problem of creating a reliable distributed system on top of less reliable edge resources. We proposed an approach to ensure timeliness in the context of tasks processing. We discussed and demonstrated that on average, our technique resulted in tasks completing with better timeliness. We proposed two mechanisms to meet per-file availability goals within the DataStore. We compared them with a static replication policy and showed that using a global availability score did the best job of meeting the availability goals.

In the future, we would like to further explore the interaction between prioritization of storage nodes to maximize performance and selecting nodes to meet availability goals. Prioritization inherently causes the load placed on the set of nodes to be unbalanced. When a node with high priority fails, a disproportionately large number of file replicas will be lost. For systems that provide data availability guarantees, accounting for this trade-off is a requirement.

Acknowledgment

The authors would like to acknowledge grant NSF CSR-1162405 and CNS-1619254 that supported this research.

References

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

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

[3] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," *IEEE pervasive Computing*, vol. 8, no. 4, 2009.

[4] V. Cunsolo, S. Distefano, A. Puliafito, and M. Scarpa, "Cloud@ home: Bridging the gap between volunteer and cloud computing," *Emerging intelligent computing technology and applications*, pp. 423–432, 2009.

[5] P. Garcia Lopez, A. Montresor, D. Epema, A. Datta, T. Higashino, A. Iamnitchi, M. Barcellos, P. Felber, and E. Riviere, "Edge-centric computing: Vision and challenges," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 5, pp. 37–42, 2015.

[6] H. P. Sajjad, K. Danniswara, A. Al-Shishtawy, and V. Vlassov, "Spanedge: Towards unifying stream processing over central and near-the-edge data centers," in *Edge Computing (SEC), IEEE/ACM Symposium on*. IEEE, 2016, pp. 168–178.

[7] A. Jonathan, M. Ryden, K. Oh, A. Chandra, and J. Weissman, "Nebula: Distributed edge cloud for data intensive computing," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2017.

[8] C. Dabrowski, "Reliability in grid computing systems," *Concurrency and Computation: Practice and Experience*, vol. 21, no. 8, pp. 927–959, 2009.

[9] K. Budati, J. Sonnek, A. Chandra, and J. Weissman, "Ridge: combining reliability and performance in open grid platforms," in *Proceedings of the 16th international symposium on High performance distributed computing*. ACM, 2007, pp. 55–64.

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

[11] G. Ananthanarayanan, S. Kandula, A. G. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, "Reining in the outliers in map-reduce clusters using mantri." in *OSDI*, vol. 10, no. 1, 2010, p. 24.

[12] T. Do, M. Hao, T. Leesatapornwongsa, T. Patana-anake, and H. S. Gunawi, "Limplock: understanding the impact of limpware on scale-out cloud systems," in *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2013, p. 14.

[13] D. Borthakur, "The hadoop distributed file system: Architecture and design," *Hadoop Project Website*, vol. 11, no. 2007, p. 21, 2007.

[14] A. Haeberlen, A. Mislove, and P. Druschel, "Glacier: Highly durable, decentralized storage despite massive correlated failures," in *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*. USENIX Association, 2005, pp. 143–158.

[15] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer *et al.*, "Oceanstore: An architecture for global-scale persistent storage," *ACM Sigplan Notices*, vol. 35, no. 11, pp. 190–201, 2000.

[16] J. S. Plank *et al.*, "A tutorial on reed-solomon coding for fault-tolerance in raid-like systems," *Softw., Pract. Exper.*, vol. 27, no. 9, pp. 995–1012, 1997.

[17] A. N. Bessani, R. Mendes, T. Oliveira, N. F. Neves, M. Correia, M. Pasin, and P. Verissimo, "Scfs: A shared cloud-backed file system." in *USENIX Annual Technical Conference*, 2014, pp. 169–180.

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

[19] Google compute engine. [Online]. Available: https://cloud.google.com/compute/

[20] A. Vulimiri, C. Curino, B. Godfrey, K. Karanasos, and G. Varghese, "Wanalytics: Analytics for a geo-distributed data-intensive world." in *CIDR*, 2015.

[21] Q. Pu, G. Ananthanarayanan, P. Bodik, S. Kandula, A. Akella, P. Bahl, and I. Stoica, "Low latency geo-distributed data analytics," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, pp. 421–434, 2015.

[22] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer, "Network-aware operator placement for stream-processing systems," in *Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on*. IEEE, 2006, pp. 49–49.

[23] K. Kloudas, M. Mamede, N. Preguiça, and R. Rodrigues, "Pixida: optimizing data parallel jobs in wide-area data analytics," *Proceedings of the VLDB Endowment*, vol. 9, no. 2, pp. 72–83, 2015.

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

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

[26] K. Oh, A. Chandra, and J. Weissman, "Wiera: Towards flexible multi-tiered geo-distributed cloud storage instances," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2016, pp. 165–176.

[27] S. Agarwal, J. Dunagan, N. Jain, S. Saroiu, A. Wolman, and H. Bhogan, "Volley: Automated data placement for geo-distributed cloud services." in *NSDI*, vol. 10, 2010, pp. 28–0.

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

[29] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *ACM SIGOPS operating systems review*, vol. 37, no. 5. ACM, 2003, pp. 29–43.

[30] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," *ACM SIGOPS operating systems review*, vol. 41, no. 6, pp. 205–220, 2007.

[31] R. Bhagwan, K. Tati, Y. Cheng, S. Savage, and G. M. Voelker, "Total recall: System support for automated availability management." in *NSDI*, vol. 4, 2004, pp. 25–25.

[32] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 4, no. 3, pp. 382–401, 1982.