Cost-efficient and Differentiated Data Availability Guarantees in Data Clouds

Nicolas Bonvin, Thanasis G. Papaioannou, Karl Aberer
School of Computer and Communication Sciences, EPFL
1015 Lausanne, Switzerland
`firstname.lastname@epfl.ch`

***Abstract*— Failures of any type are common in current datacenters. As data scales up, its availability becomes more complex, while different availability levels per application or per data item may be required. In this paper, we propose a self-managed key-value store that dynamically allocates the resources of a data cloud to several applications in a cost-efficient and fair way. Our approach offers and dynamically maintains multiple differentiated availability guarantees to each different application despite failures. We employ a virtual economy, where each data partition acts as an individual optimizer and chooses whether to migrate, replicate or remove itself based on net benefit maximization regarding the utility offered by the partition and its storage and maintenance cost. Comprehensive experimental evaluations suggest that our solution is highly scalable and adaptive to query rate variations and to resource upgrades/failures.**

I. Introduction

Cloud storage is becoming a popular business paradigm, e.g. Amazon S3, ElephantDrive, Gigaspaces, etc. The storage capacity employed may be large and it should be able to further scale up. However, as data scales up, hardware failures in current datacenters become more frequent [1]; e.g. overheating, power (PDU) failures, network failures, etc. Also, geographic proximity significantly affects data availability; e.g., in case of a PDU failure ∼500-1000 machines suddenly disappear, or in case of a rack failure ∼40-80 machines instantly go down. On the other hand, according to [2], Internet availability varies from 95% to 99.6%. Moreover, the query rates for Web applications data are highly irregular, e.g. the "Slashdot" effect. To this end, the support of service level agreements (SLAs) with data availability guarantees in cloud storage is very important. Moreover, in reality, different applications may have different availability requirements.

Skute (i.e. scattered key-value store) is designed to provide low response time on read and write operations, to ensure replicas' *geographical dispersion* in a cost-efficient way and to offer differentiated availability guarantees per data item to multiple applications, while minimizing bandwidth and storage consumption. The application data owner rents resources from a cloud of federated servers to store its data. The cloud could be a single business, i.e. a company that owns/manages data server resources, or a broker that represents servers that do not belong to the same business entities. The number of data replicas and their placement are handled by a distributed optimization algorithm autonomously executed at the servers. Also, data replication is highly adaptive to the distribution of the query load among partitions and to failures of any kind so as to maintain high data availability. Other economic-based data placement approaches are [3], [4]. However, they do not consider differentiated data availability levels, geographical distribution of replicas and different popularity of data items, as opposed to our approach.

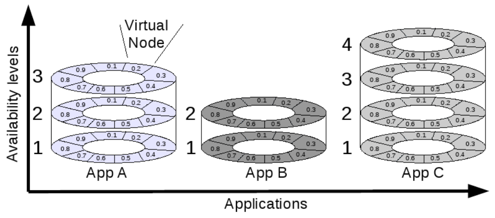

Fig. 1. 3 applications with different availability levels.

Skute is built using a ring topology and a variant of consistent hashing [5]. Data is identified by a key and its location is given by the hash function of this key, i.e. O(1) DHT. The key space is split into partitions. A physical node (i.e. server) gets assigned to multiple points in the ring, called tokens, and belongs to a rack, a room, a data center, a country and a continent. Note that a finer geographical granularity could also be considered. A virtual node (alternatively a partition) holds data for the range of keys in $(previous\ token,\ token]$, as in [5]. A virtual node may replicate or migrate its data to another server, or suicide (i.e. delete its data replica) according to the decision making process described in Section II-C. A physical node hosts a varying amount of virtual nodes depending on the query load, the size of the data managed by the virtual nodes and its own capacity (i.e. CPU, RAM, disk space, etc.).

Skute introduces the novel concept of *multiple virtual rings* on a single cloud. It allows multiple applications to share the same cloud infrastructure for offering differentiated per data item and per application availability guarantees without performance conflicts. As depicted in Fig. 1, each application uses its own virtual rings, while one ring per availability level is needed. Each virtual ring consists of multiple virtual nodes that are responsible for different data partitions of the same application that demands specific availability levels. Thus, our approach provides the following advantages over existing key-value stores:

1) *Multiple data availability levels per application.* If

one key-value store was employed per application, as suggested in [5], then an application would severely impact the performance of others that utilized the same resources. Unlike existing approaches, in Skute, every virtual node of each virtual ring acts as an individual optimizer, thus minimizing the impact among applications.

2) *Geographical data placement per application*. Data that is mostly accessed from a certain geographical region should be moved close to that region. Without the concept of virtual rings, data of different applications would have to be stored in the same partition, thus removing the option to move data close to the clients.

II. The individual optimization

Suppose that a data owner (i.e. application provider) periodically pays the operational cost of each cloud server where he stores replicas of his data partitions. The operational cost c of a server is mainly influenced by the quality of its hardware, its physical hosting, its access bandwidth, its used storage, and its query processing and communication overhead. The data owner wants to minimize his expenses by replacing expensive servers with cheaper ones, while maintaining a certain minimum data availability promised by SLAs to his clients. He also obtains some utility u from the queries answered by its data replicas that depends on the popularity of the data contained in the replica of the partition and the response time (i.e. processing and network latency) associated to the replies. The network latency depends on the distance of the clients from the server that hosts the data, i.e. the geographical distribution of query clients. Overall, a data owner seeks to maximize his aggregate net benefit, i.e. $\sum_i u_i - c_i$ for the i servers out of N that it employs for its M data partitions. This constrained global optimization problem takes $2^{M \cdot N}$ possible solutions for every application and it is feasible only for small sets of servers N and partitions M.

Instead, we employ a decentralized optimization approach. The data owner rents storage space and pays a monthly usage-based *real rent*. Each virtual node is responsible for the data in its key range and should always try to keep data availability for its partition above a certain minimum level required by the application while minimizing the associated costs (i.e. for data hosting and maintenance). To this end, a virtual node can be assumed to act as an autonomous agent on behalf of the data owner to achieve these goals. Time is assumed to be split into epochs. A virtual node pays a *virtual rent* (i.e. an approximation of the possible real rent, defined later in this section) to servers where its data is stored at each epoch. It may also replicate or migrate its data to another server, or suicide (i.e. delete its data replica). These decisions are made based on the query rate for the data of the virtual node, the renting costs and the maintenance of high availability upon failures. There is no global coordination and each virtual node behaves independently. The virtual rent of each server is announced at a board (i.e. an elected server) and is updated at the beginning of a new epoch.

A. Virtual rent

The virtual rent price c of a physical node for the next epoch is an increasing function of its query load and its storage usage at the current epoch and it can be given by:

$$c = up \cdot (1 + \alpha \cdot storage\_usage + \beta \cdot query\_load) \ , \qquad (1)$$

where up is the marginal usage price of the server, which can be calculated by the total monthly real rent paid by virtual nodes and the mean usage of the server in the previous month, and α, β are normalizing factors. We consider that the real rent price per server takes into account the network cost for communicating with the server, i.e. its access link. Access links are assumed to be the bottleneck along the path that connects any pair of servers and thus the actual distance between servers is not considered in the communication cost. Multiplying the real rent price with the query load satisfies the network proximity objective. The query load and the storage usage at the current epoch are considered to be good approximations of the ones at the next epoch, as they are not expected to change much at very small time scales. The virtual rent price per epoch is an approximation of the real monthly price that is paid by the application provider for storing the data of a virtual node. A server agent residing at the server calculates its virtual rent price per epoch.

B. Maintaining availability

A virtual node always tries to keep the data availability above a minimum level th. As estimating the probabilities of each server to fail necessitates access to an enormous set of historical data and private information of the server, we approximate the potential availability of a partition (i.e. virtual node) by means of the geographical *diversity* of the servers that host its replicas. Therefore, the availability of a partition i is defined as:

$$avail_i = \sum_{i=0}^{|S_i|} \sum_{j=i+1}^{|S_i|} conf_i \cdot conf_j \cdot diversity(s_i, s_j) \ , \qquad (2)$$

where $S_i = (s_1, s_2, \dots, s_n)$ is the set of servers hosting replicas of the virtual node i and $conf_i$, $conf_j \in [0,\ 1]$ are the confidence (i.e. a subjective estimation based on technical factors as well as non-technical factors (e.g. political and economical stability of the country, etc.)) of servers i, j. The diversity function returns a number calculated based on the geographical distance among each server pairs. This distance is represented as a 6 bit number, each bit corresponding to the location parts of a server, namely continent, country, data center, room rack and server with leftmost significance. The different location part of both servers are compared one by one to compute their *similarity*: if the location parts are equivalent, the corresponding bit is set to 1, otherwise 0. A binary "NOT" operation is then applied to the similarity to get the diversity value, e.g. $\overline{111000} = 000111 = 7(decimal)$.

When the availability of a virtual node falls below th, it replicates its data to a new server. The best candidate server is selected so as to maximize the net benefit between the diversity of the resulting set of replica locations for the virtual node and the virtual rent of the new server. Also, a preference weight g_j is associated to server j according to its location proximity to the geographical distribution G of query clients for the partition of the virtual node. Thus, the availability is increased as much as possible at the minimum cost, while the query response time is decreased. Specifically, a virtual node i with current replica locations in S_i maximizes:

$$\arg\max_j \sum_{k=1}^{|S_i|} g_j \cdot conf_j \cdot diversity(s_k, s_j) - c_j \;, \tag{3}$$

where c_j is the virtual rent price of candidate server j. g_j is calculated by a virtual node with:

$$g_j = \frac{\sum_l q_l}{1 + \sum_l q_l \cdot diversity(l, s_j)} \;, \tag{4}$$

where q_l is the number of queries for the partition of the virtual node per client location l.

C. Virtual node decision process

As already mentioned, a virtual node agent may decide to replicate, migrate, suicide or do nothing with its data at the end of an epoch. First, it verifies that the current availability of its partition satisfies the required level. If not, the virtual node replicates its data to the server that maximizes data availability at the minimum cost, as described in Subsection II-B. Upon replication, a new virtual node is associated with the replicated data. If the availability is satisfactory, the virtual node agent tries to minimize costs. During an epoch, virtual nodes receive queries, process them and send the replies back to the client. Each query creates a *utility value* for the virtual node, which can be assumed to be proportional to the size of the query reply and inversely proportional to the average distance of the client locations from the server of the virtual node. For this purpose, the balance (i.e. net benefit) b for a virtual node is defined as follows:

$$b = u(pop, g) - c \;, \tag{5}$$

where $u(pop, g)$ is assumed to be the epoch query load of the partition with popularity pop divided by the geographic proximity g (as defined in (4)) of the virtual node to the client locations and normalized to monetary units, and c is the virtual rent price. To this end, a virtual node decides to:

- *migrate or suicide:* if it has negative balance for the last f epochs. First, the virtual node calculates the availability of its partition without its own replica. If the availability is satisfactory, the virtual node suicides, i.e. deletes its replica. Otherwise, the virtual node tries (according to (3)) to find a less expensive (i.e. busy) server that is closer to the client locations.
- *replicate:* if it has positive balance b for the last f epochs, it may replicate. For replication, a virtual node has also to verify that it has enough popularity to compensate for the increased network cost for data consistency that will be paid and for the potentially increased virtual rent of the candidate server (selected according to (3)) after replication.

At the end of an epoch, the virtual node agent sets lowest utility value $u(pop, g)$ to the current lowest virtual rent price of the server to prevent unpopular nodes from migrating indefinitely. When a virtual node has a large number of queries (or queries with long responses), it becomes wealthier. At the same time, the load of the corresponding server will increase, as well as the virtual rent price for the next epoch. Popular virtual nodes on the server will have enough "money" to pay the growing rent price, as opposed to unpopular ones that will have to move to a cheaper server. The transfer of unpopular virtual nodes will in turn decrease the virtual rent price, hence stabilizing the rent price of the server. This approach is self-adaptive and balances the query load by replicating popular virtual nodes.

III. Experimental Results

A. The simulation model

We assume a simulated cloud storage environment consisting of N=200 servers geographically distributed over 10 countries (i.e. 2 datacenters/country, 1 room/datacenter, 2 racks/room, 5 servers/rack). Data (500 GB) belongs to three different applications. Each application offers one minimum availability level that is satisfied by 2, 3, 4 replicas respectively. Thus, the data of the applications 1, 2, 3 is managed by the virtual rings 0, 1, 2 respectively. At startup, data is assumed to be split into M=200 partitions per application and each partition is represented by a virtual node. We allow a maximum partition capacity of 256MB after which the data of the partition is split into two new ones. Each server has fixed and reserved bandwidth capacities of 300 MB/epoch for replication and 100 MB/epoch for migration. They also have a fixed bandwidth capacity for serving queries and a fixed storage capacity. All servers are assumed to be assigned the same confidence. The popularity of the virtual nodes (i.e. the query rate) is distributed according to Pareto(1, 50). The number of queries per epoch is Poisson distributed with a mean rate λ=3000. The geographical distribution of query clients is assumed to be Uniform and thus g_j is 1 for any server j. Time is considered to be slotted into epochs. At each epoch, virtual nodes employ the decision making algorithm of Subsection II-C. Each server updates its available bandwidth for migration, replication or answering queries, and its available storage after every data transfer that is decided to happen within one epoch. All data migrations and replications are considered to be completed as soon as decided for simplicity. The virtual price per server is determined according to (1) at the beginning of each epoch. Only network-related operational communication costs (i.e. access links) are considered significant in decision making and not the distance among servers. The real monthly operational cost c of each server is assumed to be 100$ for the 70% of the servers and 125$ for the rest.

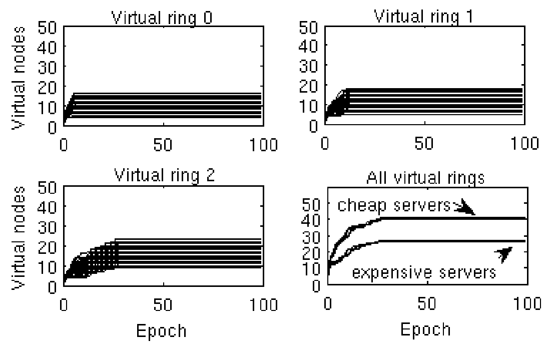

Fig. 2. Replication process at startup: the number of virtual nodes per server.

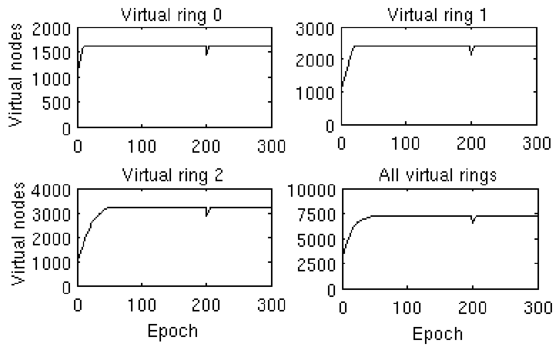

Fig. 3. Total (per ring) number of virtual nodes upon upgrades and failures.

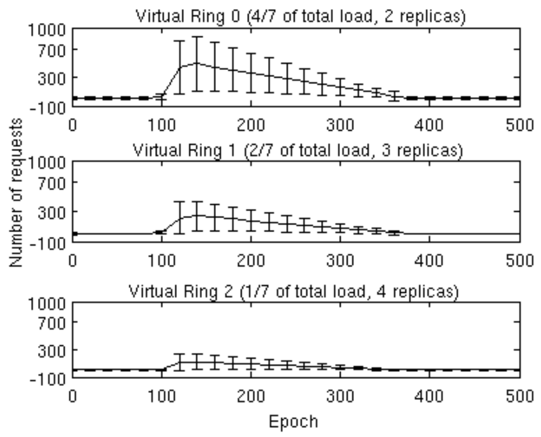

Fig. 4. Average query load per virtual ring per server over time.

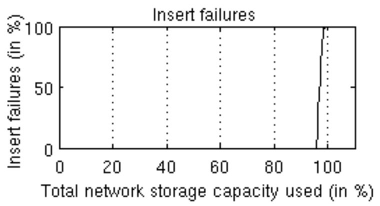

Fig. 5. Storage saturation: insert failures.

B. Convergence

As depicted in Fig. 2, the virtual nodes start replicating and migrating to other servers and the system soon reaches *equilibrium*, where fewer virtual nodes reside at expensive servers.

C. Server arrival and failure

At epoch 100, we assume that 20 new servers are added to the data cloud, while 20 different servers are removed at epoch 200. As depicted in Fig. 3, our approach is very robust to resource upgrading or failures: the total number of virtual nodes remains constant after adding resources to the data cloud and increases upon failure to maintain high availability.

D. Adaptation to the query load

Next, we simulate a load peak similar to what it would result with the Slashdot effect: At epoch 100, the mean rate queries/epoch increases from 3000 to 183000 in 25 epochs and then slowly decreases for 250 epochs until it reaches the initial rate of 3000. As shown in Fig. 4, the query load per server remains quite balanced despite the variations in the total query load, while we assume that 4/7, 2/7 and 1/7 fractions of the total query load are attracted by applications 1, 2, 3 respectively.

E. Scalability

We investigate the scalability of the approach for accomodating new data. For this purpose, we saturate the cloud capacity at a rate of 2000 insert requests/epoch (each of 500KB). These requests are Pareto(1, 50)-distributed. As depicted in Fig. 5, our approach manages to balance the used storage efficiently and fast enough so that there are no data losses for used capacity up to 96% of the total storage.

IV. Conclusion

In this paper, we proposed a decentralized economic-aware approach for replica management in order to maintain multiple availability guarantees in cloud storage. In the future, we will implement a full prototype of the approach and analyze its performance regarding latency and communication overhead.

Acknowledgment

This work was partly supported by the EU project HYDROSYS (224416, DG-INFSO).

References

[1] E. Pinheiro, W.-D. Weber, and L. A. Barroso, “Failure trends in a large disk drive population,” in *Proc. of 5th USENIX Conference on File and Storage Technologies (FAST07)*, San Jose, CA, USA, February 2007.

[2] M. Dahlin, B. B. V. Chandra, L. Gao, and A. Nayate, “End-to-end wan service availability,” *IEEE/ACM Transactions on Networking*, vol. 11, no. 2, pp. 300–313, 2003.

[3] G. Heiser, F. Lam, and S. Russell, “Resource management in the mungi single-address-space operating system,” in *Proc. of Australasian Computer Science Conference*, Perth, Australia, February 1998.

[4] R. Honicky and E. L. Miller, “A fast algorithm for online placement and reorganization of replicated data,” in *Proc. of Int. Symposium on Parallel and Distributed Processing*, Nice, France, April 2003.

[5] 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,” in *Proc. of ACM Symposium on Operating Systems Principles*, New York, NY, USA, 2007.