A Self-Organized, Fault-Tolerant and Scalable Replication Scheme for Cloud Storage

Nicolas Bonvin, Thanasis G. Papaioannou and Karl Aberer
School of Computer and Communication Sciences
École Polytechnique Fédérale de Lausanne (EPFL)
1015 Lausanne, Switzerland
Email: firstname.lastname@epfl.ch

ABSTRACT

Failures of any type are common in current datacenters, partly due to the higher scales of the data stored. 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 (i.e. a key range in a consistent-hashing space) 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. As proved by a game-theoretical model, no migrations or replications occur in the system at equilibrium, which is soon reached when the query load and the used storage are stable. Moreover, by means of extensive simulation experiments, we have proved that our approach dynamically finds the optimal resource allocation that balances the query processing overhead and satisfies the availability objectives in a cost-efficient way for different query rates and storage requirements. Finally, we have implemented a fully working prototype of our approach that clearly demonstrates its applicability in real settings.

Categories and Subject Descriptors

H.3.2 [**Information storage and retrieval**]: Information Storage; H.3.4 [**Information storage and retrieval**]: Systems and Software—*Distributed systems*; H.2.4 [**Database Management**]: Systems—*Distributed databases*; E.1 [**Data Structures**]: Distributed data structures; E.2 [**Data Storage Representations**]: Hash-table representations

General Terms

Reliability, Economics

Keywords

decentralized optimization, net benefit maximization, equilibrium, rational strategies

1. INTRODUCTION

Cloud storage is becoming a popular business paradigm, e.g. Amazon S3, ElephantDrive, Gigaspaces, etc. Small companies that offer large Web applications can avoid large capital expenditures in infrastructure by renting distributed storage and pay per use. 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 [22]; e.g. overheating, power (PDU) failures, rack failures, network failures, hard drive failures, network re-wiring and maintenance. 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. Furthermore, data may be lost due to natural disasters, such as tornadoes destroying a complete data center, or various attacks (DDoS, terrorism, etc.). On the other hand, as [7] suggests, Internet availability varies from 95% to 99.6%. Also, the query rates for Web applications data are highly irregular, e.g. the "Slashdot effect" (http://en.wikipedia.org/wiki/Slashdot_effect), and an application may become temporarily unavailable.

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. Fault-tolerance is commonly dealt with by replication. Existing works usually rely on randomness to diversify the physical servers that host the data; e.g. in [23], [17] node IDs are randomly chosen, so that peers that are adjacent in the node ID space are geographically diverse with a high probability. To the best of our knowledge, no system explicitly addresses the geographical diversity of the replicas. Also, from the application perspective, geographically distributed cloud resources have to be efficiently utilized to minimize renting costs associated to storage and communication. Clearly, geographical diversity of replica locations and communication cost are contradictory objectives. From the cloud provider perspective, efficient utilization of cloud resources is necessary both for cost-effectiveness and for accommodating load spikes. Moreover, resource utilization has to be adaptive to resource failures, addition of new resources, load variations and the distribution of client locations.

Distributed key-value store is a widely employed service case of cloud storage. Many Web applications (e.g. Amazon) and many large-scale social applications (e.g. LinkedIn, Last.fm, etc.) use distributed key-value stores. Also, several research communities (e.g. peer-to-peer, scalable distributed data structures, databases) study key-value stores, even as complete database solutions (e.g. BerkeleyDB). In this paper, we propose a scattered key-value store (referred to as "Skute"), which is designed to provide high and differentiated data availability statistical guarantees to multiple applications in a cost-efficient way in terms of rent price and query response times. A short four-pages overview of this work has been described in [5]. Our approach combines the following innovative characteristics:

- It enables a computational economy for cloud storage resources.
- It provides differentiated availability statistical guarantees to different applications despite failures by geographical diversification of replicas.
- It applies a distributed economic model for the cost-efficient self-organization of data replicas in the cloud storage that is adaptive to adding new storage, to node failures and to client locations.
- It efficiently and fairly utilizes cloud resources by performing load balancing in the cloud adaptively to the query load.

Optimal replica placement is based on distributed net benefit maximization of query response throughput minus storage as well as communication costs, under the availability constraints. The optimality of the approach is proved by comparing simulation results to those expected by numerically solving an analytical form of the global optimization problem. Also, a game-theoretic model is employed to observe the properties of the approach at *equilibrium*. A series of simulation experiments prove the aforementioned characteristics of the approach. Finally, employing a fully working prototype of Skute, we experimentally demonstrate its applicability in real settings.

The rest of the paper is organized as follows: In Section 2, the scattered key-value data store is presented. In Section 3, the global optimization problem that we address is formulated. In Section 4, we describe the individual optimization algorithm that we employ to solve the problem in a decentralized way. In Section 5, we define a game-theoretical model of the proposed mechanism and study its equilibrium properties. In Section 6, we discuss the applicability of our approach in an untrustworthy environment. In Section 7, we present our simulation results on the effectiveness of the proposed approach. In Section 8, we describe the implementation of Skute and discuss our experimental results in a real testbed. In Section 9, we outline some related work. Finally, in Section 10, we conclude our work.

2. SKUTE: SCATTERED KEY-VALUE STORE

Skute 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 ("private" clouds), or a broker that represents servers that do not belong to the same businesses ("public" clouds). 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. Defining a new approach for maintaining data consistency among replicas is not among the objectives of this work. A potential solution could be to maintain eventual data consistency among replicas by vector-clock versioning, quorum consistency mechanisms and read-repair, as in [9].

2.1 Physical node

We assume that a physical node (i.e. a server) belongs to a rack, a room, a data center, a country and a continent. Note that finer geographical granularity could also be considered. Each physical node has a label of the form "continent-country-datacenter-room-rack-server" in order to precisely identify its geographical location. For example, a possible label for a server located in a data center in Berlin could be "EU-DE-BE1-C12-R07-S34".

2.2 Virtual node

Based on the findings of [12], Skute is built using a ring topology and a variant of consistent hashing [16]. Data is identified by a key (produced by a one-way cryptographic hash function, e.g. MD5) and its location is given by the hash function of this key. The key space is split into partitions. A physical node (i.e. a server) gets assigned to multiple points in the ring, called tokens. A virtual node (alternatively a partition) holds data for the range of keys in $(previous\ token,\ token]$, as in [9]. A virtual node may replicate or migrate its data to another server, or suicide (i.e. delete its data replica) according to a decision making process described in Section 4.4. 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.).

2.3 Virtual ring

Our approach employs the concept of multiple virtual rings on a single cloud in an innovative way (*cf.* Section 9 for a comparison with [28]). Thus, as subsequently explained, we allow multiple applications to share the same cloud infrastructure for offering differentiated per data item and per application availability guarantees without performance conflicts. The single-application case with one uniform availability guarantee has been presented in [4]. In the present work, each application uses its own virtual rings, while one ring per availability level is needed, as depicted in Figure 1. Each virtual ring consists of multiple virtual nodes that are responsible for different data partitions of the same application that demand a specific availability level. This approach provides the following advantages over existing key-value stores:

1. *Multiple data availability levels per application.* Within the same application, some data may be crucial and some may be less important. In other words, an application provider may want to store data with different availability guarantees. Other approaches, such as [9], also argue that they can support several applications by deploying a key-value store per application. However, as data placement for each data store would be independent in [9], an application could severely impact the performance of others that utilize the same resources. Unlike existing approaches, Skute allows a fine-grained control of the resources of each server, as every virtual node of each virtual ring acts as an individual optimizer (as described in Section 4.4), thus minimizing the impact among applications.
2. *Geographical data placement per application.* Data that is mostly accessed from a given geographical region should be moved close to that region. Without the concept of virtual rings, if multiple applications were using the same data store,

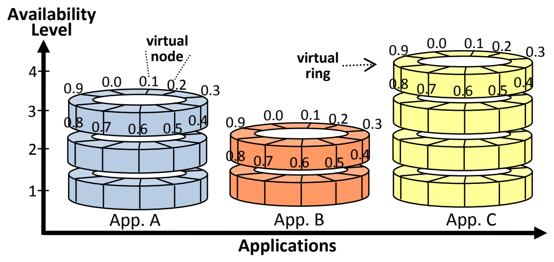

Figure 1: Three applications with different availability levels.

data of different applications would have to be stored in the same partition, thus removing the ability to move data close to the clients. However, by employing multiple virtual rings, Skute is able to provide one virtual store per application, allowing the geographical optimization of data placement.

2.4 Routing

As Skute is intended to be used with real-time applications, a query should not be routed through multiple servers before reaching its destination. Routing has to be efficient, therefore every server should have enough information in its routing table to route a query directly to its final destination. Skute could be seen as a $O(1)$ DHT, similarly to [9]. Each virtual ring has its own routing entries, resulting in potentially large routing tables. Hence, the number of entries in the routing table is:

$$entries = \sum_{i}^{apps} \sum_{j}^{levels_i} partition(i, j) \tag{1}$$

where $partition(i, j)$ returns the number of partitions (i.e. virtual nodes) of the virtual ring of the availability level i belonging to application j. However, the memory space requirement of the routing table is quite reasonable; e.g. for 100 applications, each with 3 availability levels and 5K data partitions, the necessary memory space would be ~ 31.5MB, assuming that each entry consists of 22 bytes (2 bytes for application id, 1 byte for availability level, 3 bytes for the partition id and 16 bytes for the sequence of server ids that host the partition).

A physical node is responsible to manage the routing table of all virtual rings hosted in it, in order to minimize the update costs. Upon migration, replication and suicide events, hierarchical broadcast that leverages the geographical topology of servers is employed. This approach costs $O(N)$, but it uses the minimum network spanning tree. The position of a moving virtual node (i.e. during the migration process) is tracked by forwarding pointers (e.g. SSP chains [25]). Also, the routing table is periodically updated using a gossiping protocol for shortening/repairing chains or updating stale entries (e.g. due to failures). According to this protocol, a server exchanges with random $log(N)$ other servers the routing entries of the virtual nodes that they are responsible for.

Moreover, as explained in Section 5 and experimentally proved in Section 7, no routing table updates are expected at equilibrium with stable system conditions regarding the query load and the number of servers. Even if a routing table contains a large number of entries, its practical maintenance is not costly thanks to the stability of the system. The scalability of this approach is experimentally assessed in a real testbed, as described in Section 8.

3. THE PROBLEM DEFINITION

The data belonging to an application is split into M partitions, where each partition i has r_i distributed replicas. We assume that N servers are present in the data cloud.

3.1 Maximize data availability

The first objective of a data owner d (i.e. application provider) is to provide the highest availability for a partition i, by placing all of its replicas in a set S_i^d of different servers. Data availability generally increases with the geographical diversity of the selected servers. Obviously, the worst solution in terms of data availability would be to put all replicas at a server with equal or worse probability of failure than others.

We denote as F_j a failure event at server $j \in S_i^d$. These events may be independent from each other or correlated. If we assume without loss of generality that events $F_1 \ldots F_k$ are independent and that events $F_{k+1} \ldots F_{|S_i^d|}$ are correlated, then the probability a partition i to be unavailable is given as follows:

$$\begin{aligned} &Pr(i \text{ unavailable}) = Pr(F_1 \cap F_2 \cap \ldots \cap F_{|S_i^d|}) = \\ &\prod_{j=1}^{k} Pr(F_j) \cdot Pr(F_k | F_{k+1} \ldots \cap F_{|S_i^d|}) \cdot \\ &Pr(F_{k+1} | F_{k+2} \cap \ldots \cap F_{|S_i^d|}) \cdot \ldots \cdot Pr(F_{|S_i^d|}) \,, \end{aligned} \tag{2}$$

if $F_{k+1} \cap F_{k+2} \cap \ldots F_{|S_i^d|} \neq \emptyset$.

3.2 Minimize communication cost

While geographical diversity increases availability, it is also important to take into account communication cost among servers that host different replicas, in order to save bandwidth during replication or migration, and to reduce latency in data accesses and during conflict resolution for maintaining data consistency. Let $\vec{L^d}$ be a $M \times N$ *location* matrix with its element $L_{ij}^d = 1$ if a replica of partition i of application d is stored at server j and $L_{ij}^d = 0$ otherwise. Then, we maximize data proximity by minimizing network costs for each partition i, e.g. the total communication cost for conflict resolution of replicas for the mesh network of servers where the replicas of the partition i are stored. In this case, the network cost c_n for conflict resolution of the replicas of a partition i of application d can be given by

$$c_n(\vec{L_i^d}) = sum(\vec{L_i^d} \cdot \vec{NC} \cdot \vec{L_i^d}^T) \,, \tag{3}$$

where $\vec{NC}$ is a strictly upper triangular $N \times N$ matrix whose element NC_{jk} is the communication cost between servers j and k, and sum denotes the sum of matrix elements.

3.3 Maximize net benefit

Every application provider has to periodically pay the operational cost of each server where he stores replicas of his data partitions. The operational cost of a server is mainly influenced by the quality of the hardware, its physical hosting, the access bandwidth allocated to the server, its 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 (i.e. query load) pop_i of the data contained in the replica of the partition i 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 G of query clients. Overall, he seeks to maximize his net benefit and the global optimization problem can be formulated as follows:

$$\begin{aligned}&\max\{u(pop_i, G) - \vec{L_i^d}\ \vec{c}^T + c_n(\vec{L_i^d})\}, \forall i,\ \forall d\\ &\text{s.t.}\\ &1 - Pr(F_1^{L_{i1}^d} \cap F_2^{L_{i2}^d} \cap \dots F_N^{L_{iN}^d}) \geq th_d\ ,\end{aligned} \tag{4}$$

where $\vec{c}$ is the vector of operational costs of servers with its element c_j being an increasing function of the data replicas of the various applications located at server j. This also accounts for the fact that the processing latency of a server is expected to increase with the occupancy of its resources. F_j^0 for a particular partition denotes that the partition is not present at server j and thus the corresponding failure event at this server is excluded from the intersection and th_d is a certain minimum availability threshold promised by the application provider d to his clients. 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 and partitions.

4. THE INDIVIDUAL OPTIMIZATION

The data owner rents storage space located in several data centers around the world 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 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 may replicate or migrate its data to another server, or suicide (i.e. delete its data replica) at each epoch and pay a *virtual rent* (i.e. an approximation of the possible real rent, defined later in this section) to servers where its data are stored. 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. Only one virtual node of the same partition is allowed to suicide at the same epoch by employing Paxos [18] distributed consensus algorithm among virtual nodes of the same partition. The virtual rent of each server is announced at a board and is updated at the beginning of a new epoch.

4.1 Board

At each epoch, the virtual nodes need to know the virtual rent price of the servers. One server in the network is elected (i.e. by a leader election distributed protocol) to store the current virtual rent per epoch of each server. The election is performed at startup and repeated whenever the elected server is not reachable by the majority. Servers communicate to the central board only their updated virtual prices. This centralized approach achieves common knowledge for all virtual nodes in decision making (*cf.* the algorithm of Section 4.4), but: i) it assumes trustworthiness of the elected server, ii) the elected server may become a bottleneck.

An alternative approach would be that each server maintains its own local board and periodically updates the virtual prices of a random subset ($log(N)$) of servers by contacting them directly (i.e. gossiping), having as N the total number of servers. This approach does not have the aforementioned problems, but decision making of virtual nodes is based on potentially outdated information on the virtual rents. This fully decentralized architecture has been experimentally verified in a real testbed to be very efficient without creating high communication overhead (*cf.* Section 8).

When a new server is added to the network, the data owner estimates its *confidence* based on its hardware components and its location. This estimation depends on technical factors (e.g. redundancy, security, etc.) as well as non-technicals ones (e.g. political and economical stability of the country, etc.) and it is rather subjective. Confidence values of servers are stored at the board(s) in a trustworthy setting, while they can be stored at each virtual node in case that trustworthiness is an issue. Note that asking for detailed information on the server location is already done by large companies that rent dedicated servers, e.g. theplanet.com. The potential insincerity of the server for its location could be conveyed in its confidence value based on its offered availability and performance.

4.2 Physical node

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:

$$c = up \cdot (storage\_usage + query\_load)\ , \tag{5}$$

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. 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. To this end, it is assumed that access links are going to be the bottleneck ones along the path that connects any pair of servers and thus we do not take explicitly into account distance between servers. 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 very often at very small time scales, such as a time epoch. 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. Thus, an expensive server tends to be also expensive in the virtual economy. A server agent residing at the server calculates its virtual rent price per epoch and updates the board.

4.3 Maintaining availability

A virtual node always tries to keep the data availability above a minimum level th (i.e. the availability level of the corresponding virtual ring), as specified in Section 3. 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 the sum of *diversity* of each distinct pair of servers, i.e.:

$$avail_i = \sum_{i=0}^{|S_i|} \sum_{j=i+1}^{|S_i|} conf_i \cdot conf_j \cdot diversity(s_i, s_j) \tag{6}$$

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 levels 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. Note that more bits would be required to represent additional geographical location parts than those considered. The most significant bit (leftmost) represents the continent while the least significant bit (rightmost) represents the server.

Starting with the most significant bit, each 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. Once a bit has been set to 0, all less significant bits are also set to 0. For example, two servers belonging to the same data center but located in different rooms cannot be in the same rack, thereby all bits after the third bit (data center) have to be 0. The similarity number would then look like this:

cont	coun	data	room	rack	serv
1	1	1	0	0	0

A binary "NOT" operation is then applied to the similarity to get the diversity value:

$$\overline{111000} = 000111 = 7(decimal)$$

The diversity values of server pairs are summed up, because having more replicas in distinct servers located even in the same location always results in increased availability.

When the availability of a virtual node falls below th, it replicates its data to a new server. Note that a virtual node can know the locations of the replicas of its partition by the routing table of its hosting server and thus calculate its availability according to 6. 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 is associated to servers according to their location proximity to the geographical distribution G of query clients for the partition. G is approximated by the virtual node by storing the number of client queries per location. Thus, the availability is increased as much as possible at the minimum cost, while the network latency for the query reply is decreased. Specifically, a virtual node i with current replica locations in S_i maximizes:

$$\arg\max_{j} \sum_{k=1}^{|S_i|} g_j \cdot conf_j \cdot diversity(s_k, s_j) - c_j \ , \qquad (7)$$

where c_j is the virtual rent price of candidate server j. g_j is a weight related to the proximity (i.e. inverse average diversity) of the server location to the geographical distribution of query clients for the partition of a virtual node and is given by:

$$g_j = \frac{\sum_l q_l}{1 + \sum_l q_l \cdot diversity(l, s_j)} \ , \qquad (8)$$

where q_l is the number of queries for the partition of the virtual node per client location l. To this end, we assume that the client locations are encoded similarly to those of servers. In fact, if client requests reach the cloud by the geographically nearest cloud node to the client (e.g. by employing geoDNS), we can take the location of this cloud node as the client location. However, having virtual nodes to choose the destination server j for replication according to (7) would render j a bottleneck for the next epoch. Instead, the destination server is randomly chosen among the top-k ones that are ranked according to the maximized quantity in (7).

The minimum availability level th allows a fine-grained control over the replication process. A low value means that a partition will be replicated on few servers potentially geographically close, whereas a higher value enforces many replicas to be located at dispersed locations. However, setting a high value for the minimum level of availability in a network with a few servers can result in an undesirable situation, where all partitions are replicated everywhere. To circumvent this, a maximum number of replicas per virtual node is allowed.

4.4 Virtual node decision tree

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. Note that decision making of virtual nodes does not need to be synchronized. Upon replication, a new virtual node is associated with the replicated data. The decision tree of a virtual node is depicted in Figure 2. First, it verifies that the current availability of its partition is greater than th. Note If the minimum acceptable availability is not reached, the virtual node replicates its data to the server that maximizes availability at the minimum cost, as described in Subsection 4.3.

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:

$$b = u(pop, g) - c \ , \qquad (9)$$

where $u(pop, g)$ is assumed to be the epoch query load of the partition with a certain popularity *pop* divided by the proximity g (as defined in (8)) 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 to find a less expensive (i.e. busy) server that is closer to the client locations (according to maximization formula (7)). To avoid a data replica oscillation among servers, the migration is only allowed if the following *migration conditions* apply:
  - The minimum availability is still satisfied using the new server,
  - the absolute price difference between the current and the new server is greater than a threshold,
  - the current server storage usage is above a storage soft limit, typically 70% of the hard drive capacity, and the new server is below that limit.
- *replicate:* if it has positive balance for the last f epochs, it may replicate. For replication, a virtual node has also to verify that:
  - It can afford the replication by having a positive balance b' for consecutive f epochs:

    $$b' = u(pop, g) - c_n - 1.2 \cdot c'$$

    where c_n is a term representing the consistency (i.e. network) cost, which can be approximated as the number of replicas of the partition times a fixed average communication cost parameter for conflict resolution and routing table maintenance. c' is the current virtual rent of the candidate server for replication (randomly selected among the top-k ones ranked according to the formula (7)), while the factor 1.2 accounts for an upper bounded 20% increase at this rent price at the next epoch due to the potentially increased occupied storage and query load of the candidate server. This action aims to distribute to load of the current server towards one located closer to the clients. Thus, it tends to decrease the processing and network latency of the queries for the partition.

– the average bandwidth consumption bdw_r for the answering queries per replica after replication (left term of left side of inequality (10)) plus the size p_s of the partition is less than the respective bandwidth bdw per replica without replication (right side of inequality (10)) for a fixed number win of epochs to compensate for steep changes of the query rate. A large win value should be used for bursty query load. Specifically, the virtual node replicates if:

$$\frac{win * q * q_s}{|S_i| + 1} + p_s < \frac{win * q * q_s}{|S_i|}, \qquad (10)$$

where q is the average number of queries for the last win epochs, q_s is the average size of the replies, $|S_i|$ is the number of servers currently hosting replicas of partition i and p_s is the size of the partition.

At the end of a time 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.

A virtual node that either gets a large number of queries or has to provide large query replies becomes wealthier. At the same time, the load of its hosting 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.

5. EQUILIBRIUM ANALYSIS

First, we find the expected payoffs of the actions of a virtual node, as described in our mechanism. Assume that M is the original number of partitions (i.e. virtual nodes) in the system. These virtual nodes may belong to the same or to different applications and compete to each other. Time is assumed to be slotted in rounds. At each round, a virtual node (which is either responsible for an original partition or a replica of the partition) is able to migrate, replicate, suicide (i.e. delete itself) or do nothing (i.e. stay) competitively to other virtual nodes at a repeated game. The expected single round strategy payoffs at round $t + 1$ by the various actions made at round t of the game for a virtual node i are given by:

- *Migrate:* $EV_M = \frac{u_i^{(t)}}{r_i^{(t)}} - f_c^i - f_d^i \cdot r_i^{(t)} - C_c^{(t+1)}$
- *Replicate:* $EV_R = \frac{u_i^{(t)}+a_i^{(t)}}{r_i^{(t)}+1} - f_c^i - f_d^i(r_i^{(t)}+1) - \frac{1}{2}(C_c^{(t+1)} + C_e^{(t+1)})$
- *Suicide:* $EV_D = 0$
- *Stay:* $EV_S = \frac{u_i^{(t)}}{r_i^{(t)}} - f_d^i r_i^{(t)} - C_e^{(t+1)}$

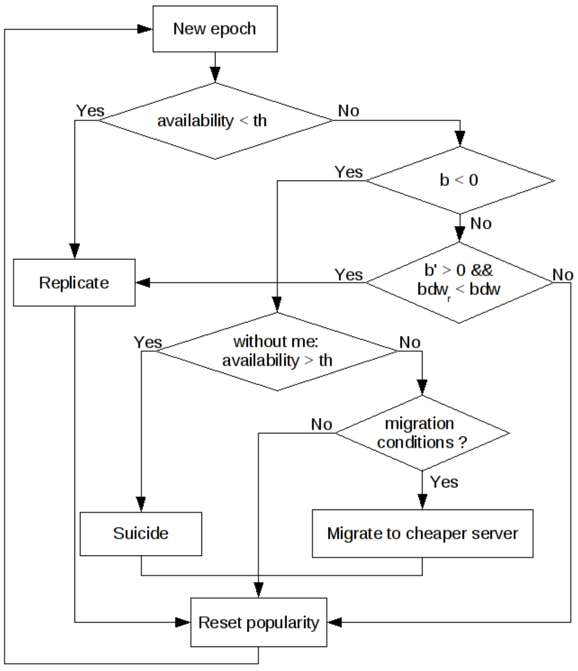

Figure 2: Decision tree of the virtual node agent.

$u_i^{(t)}$ is the utility gained by the queries served by the partition for which virtual node i is responsible and only depends on its popularity at round t; for simplicity and without loss of generality, we assume that clients are geographically uniformly distributed. To this end, a virtual node expects that this popularity will be maintained at the next round of the game. $r_i^{(t)}$ is the number of replicas of virtual node i at round t. $C_c^{(t+1)}$ is the expected price at round $t + 1$ of the cheapest server at round t. Note that it is a dominant strategy for each virtual node to select the cheapest server to migrate or replicate to, as any other choice could be exploited by competitive rational virtual nodes. f_d^i is the mean communication cost per replica of virtual node i for data consistency, f_c^i is the communication cost for migrating virtual node i, $a_i^{(t)}$ is the utility gain due to the increased availability of virtual node i when a new replica is created, and $C_e^{(t+1)}$ is the price at round $t + 1$ of the current hosting server at round t, therefore $C_e^{(t)} > C_c^{(t)}$. In case of replication, two virtual nodes will henceforth exist in the system, having equal expected utilities, but the old one paying $C_e^{(t+1)}$ and the new one paying $C_c^{(t+1)}$. In the aforementioned formula of EV_R, we calculate the expected payoff per copy of the virtual node after replication.

Notice that EV_R is expected to be initially significantly above 0, as the initial utility gain a from availability should be large in order the necessary replicas to be created. Also, if the virtual price difference among servers is initially significant, then $EV_M - EV_S$ will be frequently positive and virtual nodes will migrate towards cheaper servers. As the number of replicas increases, a decreases (and eventually becomes a small constant close to 0 after the required availability is reached) and thus EV_R decreases. Also, as price difference in the system is gradually balanced, the difference $EV_M - EV_S$ becomes more frequently negative, so fewer migrations happen. On the other hand, if the popularity (i.e. query load) of a virtual node is significantly deteriorated (i.e. u decreases), while its minimum availability is satisfied (i.e. a is close to 0), then it may become preferable for a virtual node to commit suicide.

Next, we consider the system at equilibrium and that the system conditions, namely the popularity of virtual nodes and the number of servers, remain stable. If we assume that each virtual node i plays a mixed strategy among his pure strategies, specifically it plays migrate, replicate, suicide and stay with probabilities x, y, z and $1-x-y-z$ respectively, then we calculate $C_c^{(t+1)}$, $C_e^{(t+1)}$ as follows:

$$C_c^{(t+1)} = C_c^{(t)}[1 + (x+y)\sum_{i=1}^{M} r_i^{(t)}] \quad (11)$$

$$C_e^{(t+1)} = C_e^{(t)}[1 - (x + z + \phi y)] \quad (12)$$

In equation (11), we assume that the price of the cheapest server at the next time slot increases linearly to the number of replicas that are expected to have migrated or replicated to that server until the next time slot. Also, in equation (12), we assume that the expected price of the current server at the next time slot decreases linearly to the fraction of replicas that are expected to abandon this server or replicate until the next time slot. $0 < \phi << 1$ is explained as follows: Recall that the total number of queries for a partition is divided by the total number of replicas of that partition and thus replication also reduces the rent price of the current server. However, the storage cost for hosting the virtual node remains and, as the replicas of the virtual node in the system increase, it becomes the dominant cost factor of the rent price of the current server. Therefore, replication only contributes to $C_e^{(t+1)}$ in a limited way, as shown in equation (12). Note that any cost function (e.g. a convex one, as storage is a constrained resource) could be used in our equilibrium analysis, as long as it was increasing to the number of replicas, which is a safe assumption.

Henceforth, for simplicity, we drop i indices as we deal only with one virtual node. Recall that the term a becomes close to 0 at equilibrium. Then, the replicate strategy is dominated by the migrate one, and thus $y = 0$. Also, the suicide strategy has to be eventually dominated by the migrate and stay strategies, because otherwise every virtual node would have have the incentive to leave the system; thus $z = 0$. Therefore, the number r of replicas of a virtual node becomes fixed at equilibrium and the total sum N_r of the replicas of all virtual nodes in the cloud is also fixed. As $y = z = 0$ at equilibrium, the virtual node plays a mixed strategy among migrate and stay with probabilities x and $1-x$ respectively. The expected payoffs of these strategies should be equal at equilibrium, as the virtual node should be indifferent between them:

$$\begin{aligned} &EV_M = EV_S \Leftrightarrow \\ &\frac{u}{r} - f_c - f_d\, r - C_c(1 + x\, N_r) = \frac{u}{r} - f_d\, r - C_e(1-x) \Leftrightarrow \\ &x = \frac{C_e - C_c - f_c}{C_e + C_c\, N_r} \end{aligned} \quad (13)$$

The nominator of x says that in order for any migrations to happen in the system at equilibrium the rent of the current server used by a virtual node should exceed the rent of the cheapest server more than the cost of migration for this virtual node. Also, the probability to migrate decreases with the total number of replicas in the system. Considering that each migration decreases the average virtual price difference in the system, then the number of migrations at equilibrium will be almost 0.

6. RATIONAL STRATEGIES

We have already accounted for the case that virtual nodes are rational, as we have considered them to be individual optimizers. In this section, we consider the rational strategies that could be employed by servers in an untrustworthy environment. No malicious strategies are considered, such as tampering with data, data deliberate destruction or theft, because standard cryptographic methods (e.g. digital signatures, digital hashes, symmetric encryption keys) could easily alleviate them (at a performance cost) and the servers would have legal consequences if discovered employing them. Such cryptographic methods should be employed in a real untrustworthy environment, but we refrain from further dealing with them in this paper. However, rational servers could have the incentive to lie about their virtual prices, so that they do not reflect the actual usage of their storage and bandwidth resources. For example, a server may overutilize its bandwidth resources by advertising a lower virtual price (or equivalently a lower bandwidth utilization) than the true one and increase its profits by being paid by more virtual nodes. At this point, recall that the application provider pays a monthly rent per virtual node to each server that hosts its virtual nodes. In case of server overutilization, some queries to the virtual nodes of the server would have to be buffered or even dropped by the server. Also, one may argue that a server can increase its marginal usage price on will in this environment, which then is used to calculate the monthly rent of a virtual node. This is partly true, despite competition among servers, as the total actual resource usage of a server per month cannot be easily estimated by individual application providers.

The aforementioned rational strategies could be tackled as follows: In Section 4, we assumed that virtual nodes assign to servers a subjective confidence value based on the quality of the resources of the servers and their location. In an untrustworthy environment, the confidence value of a server could also reflect its *trustworthiness* for reporting its utilization correctly. This trustworthiness value could be effectively approximated by the application provider by means of reputation based on periodical monitoring of the performance of servers to own queries. The aforementioned rational strategies are common in everyday transactions among sellers and buyers, but in a competitive environment, comparing servers based on their prices and their offered performance provides them with the right incentives for truthful reporting [10]. Therefore, in a cloud with rational servers, application providers should divide c_j by the confidence $conf_j$ of the server j in the maximization formula (7), in order to provide incentives to servers to refrain from employing the aforementioned strategies.

7. SIMULATION RESULTS

7.1 The simulation model

We assume a simulated cloud storage environment consisting of N servers geographically distributed according to different scenarios that are explained on a per case basis. Data per application is assumed to be split into M partitions having each represented by a virtual node. Each server has fixed bandwidth capacities for replication and migration per epoch. 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, which is different per experiment. For facilitating the comparison of the simulation results with those of the analytical model of Section 3, the geographical distribution of query clients is assumed to be Uniform and thus g_j is 1 for any server j. The size of every data partition is assumed to be fixed and equal to 256MB. Time is considered to be slotted into epochs. At each epoch, virtual nodes employ the decision making algorithm of

Subsection 4.4. Note that decision making of virtual nodes is not synchronized. 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. Previous data migrations and replications are taken into account in the next epoch. The virtual price per server is determined according to formula (5) at the beginning of each epoch.

7.2 Convergence to equilibrium and optimal solution

We first consider a *small scale* scenario to validate our results solving numerically the optimization problem of Section 3. Specifically, we consider a data cloud consisting of $N = 5$ servers dispersed in Europe: two servers are hosted in Switzerland in separate data centers, one in France and two servers are hosted in Germany in the same rack of the same data center. Data belongs to two applications and it is split into $M = 50$ partitions per application that are randomly shared among servers at startup. The mean query rate is $\lambda = 300$ queries per epoch. The minimum availability level in the simulation model is configured so as to ensure that each partition of the first (resp. second) application is hosted by at least 2 (resp. 4) servers located at different data centers. In the analytical model of Section 3, we assume that each server has probability 0.3 to fail and that the failure probabilities of the first 3 server are independent, while those of the Germany data centers are correlated, so as $Pr[F_4|F_5] = Pr[F_5|F_4] = 0.5$. We set $th_1 = 0.9$ for the first application and $th_2 = 0.985$ for the second application. Only network-related operational costs (i.e. access links) are considered the dominant factor for the communication cost and thus distance of servers is not taken into account in decision making; therefore we assume $c_n = 0$, in both the simulation and the analytical model. The same confidence is assigned to all servers in the simulation model. The monthly operational cost c of each server is assumed to be 100\$. Also, as the geographical distribution of query clients is assumed to be Uniform, the utility function in the analytical model only depends on the popularity pop_i of the virtual node i and is taken equal to $100 \cdot pop_i$. The detailed parameters of this experiment are shown in the left column (small scale) of Table 1.

Table 1: Parameters of small-scale and large-scale experiments.

Parameter	Small scale	Large scale
Servers	5	200
Server storage	10 GB	10 GB
Server price	100$	100$ (70%), 125$ (30%)
Total data	10 GB	100 GB
Average size of an item	500 KB	500 KB
Partitions	50	10000
Queries per epoch	Poisson ($\lambda = 300$)	Poisson ($\lambda = 3000$)
Query key distribution	Pareto (1,50)	Pareto (1,50)
Storage soft limit	0.7	0.7
Win	20	100
Replication bandwidth	300 MB/epoch	300 MB/epoch
Migration bandwidth	100 MB/epoch	100 MB/epoch

As depicted in Figure 3, the virtual nodes start replicating and migrating to other servers and the system soon reaches *equilibrium*, as predicted in Section 5. The convergence process actually takes only about 8 epochs, which is very close to the communication bound for replication (i.e. total data size / replication bandwidth = 10GB / 1.5GB per epoch$\approx$ 6.6 epochs). Also, as revealed by comparing the numerical solution of the optimization problem of Section 3 with the one that is given by simulation experiments, the proposed distributed economic approach solves rather accurately the optimization problem. Specifically, the average number of virtual nodes of either application per server were the same and the distributions of virtual nodes of either application per server were similar.

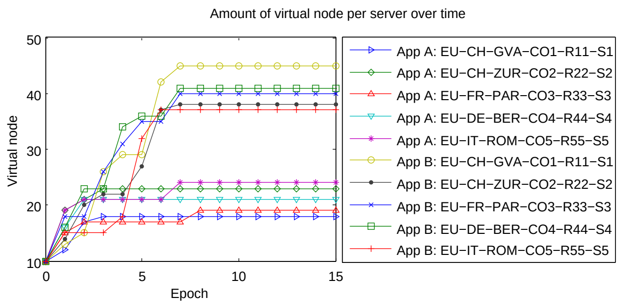

Figure 3: Small-scale scenario: replication process at startup.

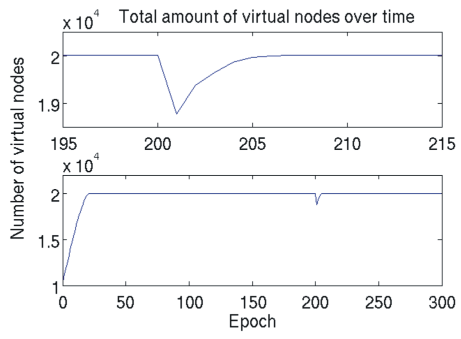

Figure 4: Large-scale scenario: robustness against upgrades and failures.

7.3 Server arrival and failure

Henceforth, we consider a more realistic *large-scale* scenario of $M = 10000$ partitions and $N = 200$ servers of different real rents. Now, data belongs to three different applications. The desired availability levels for applications 1, 2, 3 pose a requirement for a minimum number of 2, 3, 4 replicas respectively in the data store. One virtual ring is employed per application. Servers are shared among 10 countries with 2 datacenters per country, 1 room per datacenter, 2 racks per room, and 5 servers per rack. The other parameters of this experiment are shown in the right column (large-scale) of Table 1. At epoch 100, we assume that 30 new servers are added to the data cloud, while 30 random servers are removed at epoch 200. As depicted in Figure 4, 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. Note that the average number of virtual nodes per server decreases after resource upgrading, as the same total number of virtual nodes is shared among a larger number of servers.

7.4 Adaptation to the query load

Next, in order to show the adaptability of the store to the query load, we simulate a load peak similar to what it would result with the “Slashdot effect”: in a short period the query rate gets multiplied by 60. Hence, at epoch 100 the mean rate of queries per epoch increases from 3000 to 183000 in 25 epochs and then slowly decreases for 250 epochs until it reaches the normal rate of 3000

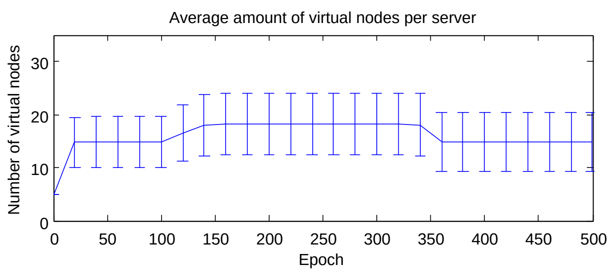

Figure 5: Large-scale scenario: total amount of virtual nodes in the system over time.

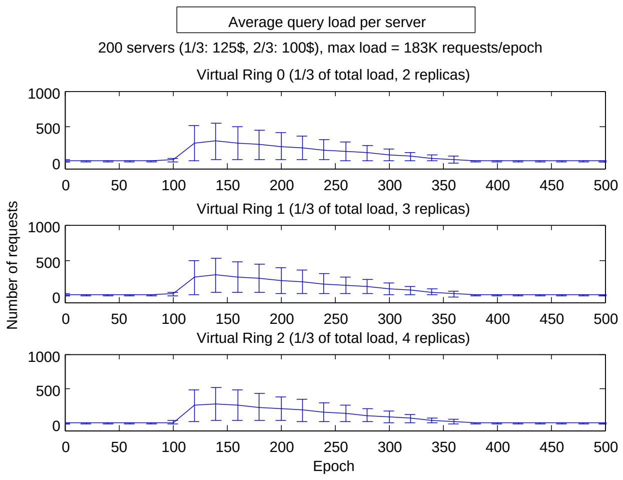

Figure 6: Large scale scenario: average query load per virtual ring per server over time when the queries are evenly distributed among applications.

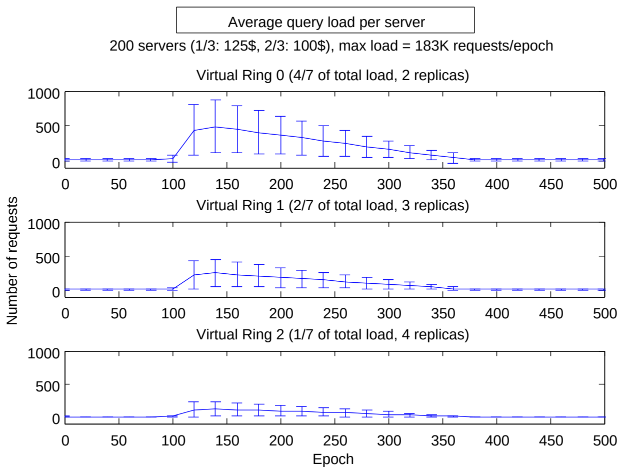

Figure 7: Large-scale scenario: average query load per virtual ring per server over time when 4/7, 2/7, 1/7 of the queries are attracted by application 1, 2, 3 respectively.

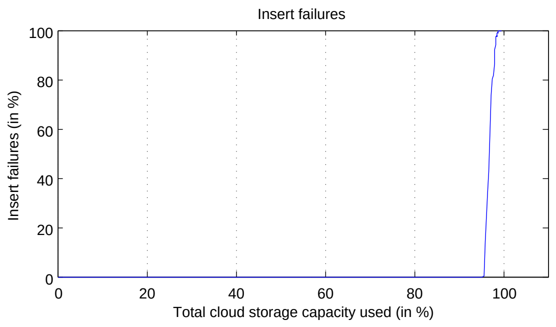

Figure 8: Storage saturation: insert failures

queries per epoch. The other parameters of this experiment are those of the large-scale scenario of Table 1. Following the Pareto distribution properties, a small amount of virtual nodes are responsible for a large amount of queries. These virtual nodes become wealthier thanks to their high popularity, and they are able to replicate to one or several servers in order to handle the increasing load. Therefore, the total amount of virtual nodes is adjusted to the query load, as depicted in Figure 5. The number of virtual nodes remains almost constant during the high query load period. This is explained as follows: For robustness, replication is only initiated by a high query load. However, a replicated virtual node can survive even with a small number of requests before committing suicide. Therefore, the number of virtual nodes decreases when the query load is significantly reduced. Finally, at epoch 375, the balance of the additional replicated virtual nodes becomes negative and they commit suicide. More importantly, the query load per server remains quite balanced despite the variations in the total query load. This is true both for the case that the query load is evenly distributed among applications (see Figure 6) and for the case that 4/7, 2/7 and 1/7 fractions of the total query load are attracted by application 1 (virtual ring 0), 2 (virtual ring 1) and 3 (virtual ring 2) respectively (see Figure 7).

7.5 Scalability of the approach

Initially, we investigate the scalability of the approach regarding the storage capacity. For this purpose, we assume the arrival of insert queries that store new data into the cloud. The insert queries are again distributed according to Pareto(1, 50). We allow a maximum partition capacity of 256MB after which the data of the partition is split into two new ones, so that each virtual node is always responsible for up to 256MB of data. The insert query rate is fixed and equal to 2000 queries per epoch, while each query inserts 500KB of data. We employ the large-scale scenario parameters, but with the number of servers $N = 100$ and 2 racks per room in this case. The initial number of partitions is $M = 200$. We fill the cloud up to its total storage capacity. As depicted in Figure 8, 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. At that point, virtual nodes start not fitting to the available storage of the individual servers and thus they cannot migrate to accommodate their data.

Next, we consider that the query rate to the cloud is not distributed according to Poisson, but it increases with the rate of 200 queries per epoch until the total bandwidth capacity of the cloud is saturated. In this experiment, real rents of servers are uniformly distributed in [1, 100]$. Now, our approach for selecting the desti-

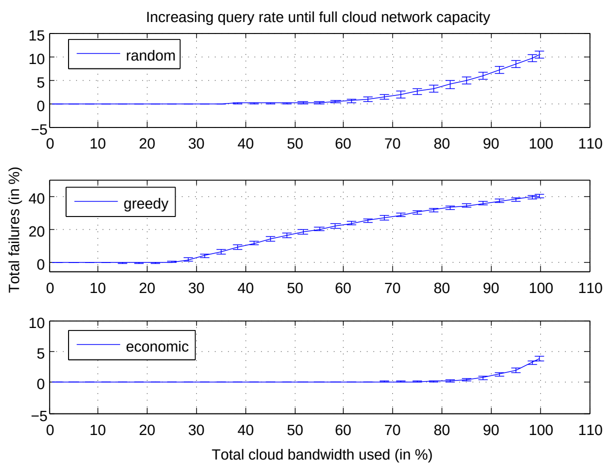

Figure 9: Network saturation: query failures

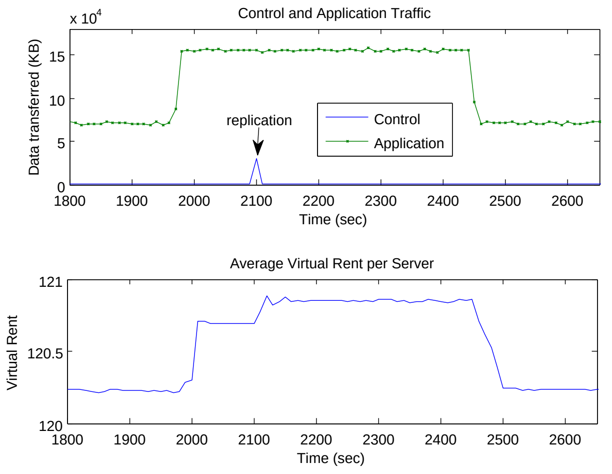

Figure 10: Top: Application and control traffic in case of a load peak. Bottom: Average virtual rent in case of a load peak.

nation server of a new replica is compared against two other rather basic approaches:

- *Random*: a random server is selected for replication and migration, as long as it has the available bandwidth capacity for migration and replication, and enough storage space.
- *Greedy*: the cheapest server is selected for replication and migration, as long as it has the available bandwidth capacity for migration and replication, and enough storage space.

As depicted in Figure 9, our approach (referred to as "economic") outperforms the simple approaches regarding the amount of dropped queries having the bandwidth of the cloud completely saturated. Specifically, only 5% of the total queries are dropped at this worst case scenario. Therefore, our approach multiplexes the resources of the cloud very efficiently.

8. IMPLEMENTATION AND EXPERIMENTAL RESULTS IN A REAL TESTBED

We have implemented a fully working prototype of Skute on top of Project Voldemort (project-voldemort.com), which is an open source implementation of Dynamo [9] written in Java. Servers are not synchronized and no centralized component is required. The epoch is considered to be equal to 30 seconds. We have implemented a fully decentralized board based on a gossiping protocol, where each server exchanges its virtual rent price periodically with a small ($log(N)$, where N is the total number of servers) random subset of servers. Routing tables are maintained using a similar gossiping protocol for routing entries. The periods of these gossiping protocols are assumed to be 1 epoch. In case of migration, replication or suicide of a virtual node, the hosting server broadcasts the routing table update using a distribution tree leveraging the geographical topology of the servers.

Our testbed consists of $N = 40$ Skute servers, hosted by 8 machines (OS: Debian 5.0.3, Kernel: 2.6.26-2-amd64, CPU: 8 core Intel Xeon CPU E5430 @ 2.66GHz, RAM: 16GB) with Sun Java 64-Bit VMs (build 1.6.0_12-b04) and connected in a 100 Mbps LAN. According to our scenario, we assume a Skute data cloud spanning across 4 European countries with 2 datacenters per country. Each datacenter is hosted by a separate machine and contains 5 Skute servers, which are considered to be at the same rack. We consider 3 applications, each of $M = 50$ partitions, with a minimum required availability of 2, 3 and 4 replicas respectively. 250000 data items of 10KB have been evenly inserted in the 3 applications. We generate 100 data requests per second using a Pareto(1,50) key distribution, denoted as application traffic. We refer as control traffic to the data volume transferred for migrations, replications and the maintenance of the boards as well as the routing tables.

We first evaluate the behavior of the system in case of a load peak. At second 1980, additional 100 requests per second are generated for a unique key. After 100 seconds, at second 2080, the popular virtual node hosting this unique key is replicated, as shown by the peak in the control traffic in Figure 10(top). Moreover, as depicted in Figure 10(bottom), the average virtual rent price increases during the load peak, as more physical resources are required to serve the increased number of requests. It further increases after the replication of the popular virtual node, because more storage is used at a server for hosting the new replica of the popular partition.

Next, the behavior of the system in case of a server crash is assessed. At second 2800, a Skute server collapses. As soon as the virtual nodes detect the failure (by means of the gossiping protocols), they start replicating the partitions hosted on the failed Skute server to satisfy again the minimum availability guarantees. Figure 11(top) shows that the replication process (as revealed by the increased control traffic) starts directly after the crash. Moreover, as depicted in Figure 11(bottom), the average virtual rent increases during the replication process, because the same storage and processing requirements as before the crash, have to be now satisfied by fewer servers.

Finally, note that in every case and especially when the system is at equilibrium the control traffic is minimal as compared to the application one.

9. RELATED WORK

Dealing with network failure, strong consistency (which databases care of) and high data availability can not be achieved at the same time [3]. High data availability by means of replication has been investigated in various contexts, such as P2P systems [23, 17], data clouds, distributed databases [21, 9] and distributed file systems [13, 24, 1, 11]. In the P2P storage systems PAST [23] and

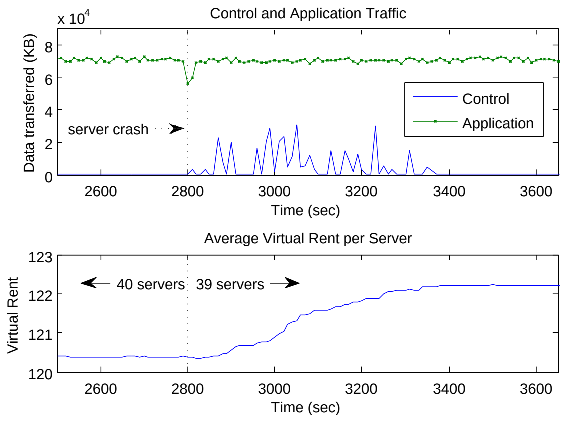

Figure 11: Top: Application and control traffic in case of a server crash. Bottom: Average virtual rent in case of a server crash.

Oceanstore [17], the geographical diversity of the replicas is based on random hashing of data keys. Oceanstore deals with consistency by serializing updates on replicas and then applying them atomically. In the distributed databases and systems context, Coda [24], Bayou [21] and Ficus [13] allow disconnected operations and are resilient to issues, such as network partitions and outages. Conflicts among replicas are dealt with different approaches that guarantee event causality. In distributed data clouds, Amazon Dynamo [9] replicates each data item at a fixed number of physically distinct nodes. Dynamo deals with load balancing by assuming the uniform distribution of popular data items among nodes through partitioning. However, load balancing based on dynamic changes of query load are not considered. Data consistency is handled based on vector clocks and a quorum system approach with a coordinator for each data key. In all the aforementioned systems, replication is employed in a static way, i.e. the number of replicas and their location are predetermined. Also, no replication cost considerations are taken into account and no geographical diversity of replicas is employed.

In [28], data replicas are organized in multiple rings to achieve query load-balancing. However, only one ring is materialized (i.e. has a routing table) and the other rings are accessible by iteratively applying a static hash function. This static approach for mapping replicas to servers does not allow to perform advanced optimizations, such as moving data close to the end user or ensuring the geographical diversity between replicas. Moreover, as opposed to our approach, the system in [28] does not support a different availability level per application or per data item, while the data belonging to different applications is not separately stored.

Some economic-aware approaches are dealing with the optimal locations of replicas. Mariposa [27] aims at latency minimization in executing complex queries over relational distributed databases, i.e. not primary-key access queries on which we focus. Sites in Mariposa exchange data items (i.e. migrate or replicate them) based on their expected query rate and their processing cost. The data items are exchanged based on their expected values using combinatorial auctions, where winner determination is tricky and synchronization is required. In our approach, asynchronous individual decisions are taken by data items regarding replication, migration or deletion, so that high availability is preserved and dynamic load balancing is performed. Also, in [26], a cost model is defined for the factors that affect data and application migration for minimizing latency in replying queries. Data is migrated towards the application or the application towards the data based on their respective costs that depends on various aspects, such as query load, replicas placement and network and storage availability.

On the other hand, in the Mungi operating system [14], a commodity market of storage space has been proposed. Specifically, storage space is lent by storage servers to users and the rental prices increase as the available storage runs low, forcing users to release unneeded storage. This model is equivalent to that of dynamic pricing per volume in telecommunication networks according to which prices increase with the level of congestion, i.e. congestion pricing. Occupied storage is associated to specific objects that are linked to bank accounts from which rent is collected for the storage. This approach does not take into account the different query rates for the various data items and it does not have any availability objectives.

In [15], an approach is proposed for reorganizing replicas evenly in case that new storage is added into the cloud, while minimizing data movement. Relocated data and new replicas are assigned with higher probability to newer servers. Replication process randomly determines the locations of replicas, while preserving that no replicas are placed in the same server. However, this approach does not consider geographical distribution of replicas or differentiated availability levels to multiple applications, and it does not take into account popularity of data items in the replication process.

In [19], an approach has been proposed for optimally selecting the query plan to be executed in the cloud in a cost-efficient way considering the load of remote servers, the latency among servers and the availability of servers. This approach has similar objectives to ours, but the focus of our paper is solely on primary-key queries.

In [2] and [8] efficient data management for the consistency of replicated data in distributed databases is addressed by an approach guaranteeing one-copy serializability in the former and snapshot isolation in lazy replicated databases (i.e. where replicas are synchronized by separate transactions) in the latter. In our case, we do not expect high update rates in a key-value store and therefore concurrent copy of changes to all replicas can be an acceptable approach. However, regarding fault tolerance against failures during updates, the approach of [2] could be employed, so as the replicas of a partition to be organized in a tree.

10. CONCLUSION

In this paper, we described Skute, a robust, scalable and highly-available key-value store that dynamically adapts to varying query load or disasters by determining the most cost-efficient locations of data replicas with respect to their popularity and their client locations. We experimentally proved that our approach converges fast to equilibrium, where as predicted by a game-theoretical model no migrations happen for steady system conditions. Our approach achieves net benefit maximization for application providers and therefore it is highly applicable to real business cases. We have built a fully working prototype in a distributed setting that clearly demonstrates the feasibility, the effectiveness and the low communication overhead of our approach. As a future work, we plan to investigate the employment of our approach for more complex data models, such as the one in Bigtable [6].

11. ACKNOWLEDGMENTS

This work was partly supported by the EU projects HYDROSYS (224416, DG-INFSO) and OKKAM (215032, ICT).

12. REFERENCES

[1] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. Farsite: federated, available, and reliable storage for an incompletely trusted environment. *ACM SIGOPS Operating Systems Review*, 36(SI):1–14, 2002.

[2] D. Agrawal and A. E. Abbadi. The tree quorum protocol: An efficient approach for managing replicated data. In *VLDB '90: Proc. of the 16th International Conference on Very Large Data Bases*, pages 243–254, Brisbane, Queensland, Australia, 1990.

[3] P. A. Bernstein and N. Goodman. An algorithm for concurrency control and recovery in replicated distributed databases. *ACM Transactions on Database Systems*, 9(4):596–615, 1984.

[4] N. Bonvin, T. G. Papaioannou, and K. Aberer. Dynamic cost-efficient replication in data clouds. In *Proc. of the Workshop on Automated Control for Datacenters and Clouds*, Barcelona, Spain, June 2009.

[5] N. Bonvin, T. G. Papaioannou, and K. Aberer. Cost-efficient and differentiated data availability guarantees in data clouds. In *ICDE '10: Proceedings of the 26th IEEE International Conference on Data Engineering*, Long Beach, CA, USA, March 2010.

[6] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. In *Proc. of the Symposium on Operating Systems Design and Implementation*, Seattle, Washington, 2006.

[7] M. Dahlin, B. B. V. Chandra, L. Gao, and A. Nayate. End-to-end wan service availability. *IEEE/ACM Transactions on Networking*, 11(2):300–313, 2003.

[8] K. Daudjee and K. Salem. Lazy database replication with snapshot isolation. In *VLDB '06: Proc. of the 32nd International Conference on Very large data bases*, pages 715–726, Seoul, Korea, 2006.

[9] 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.

[10] C. Dellarocas. Goodwill hunting: An economically efficient online feedback mechanism for environments with variable product quality. In *Proc. of the Workshop on Agent-Mediated Electronic Commerce*, July 2002.

[11] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proc. of Symposium on Operating Systems Principles*, pages 29–43, Bolton Landing, NY, USA, 2003.

[12] R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica. The impact of dht routing geometry on resilience and proximity, 2003.

[13] R. G. Guy, J. S. Heidemann, and J. T. W. Page. The ficus replicated file system. *ACM SIGOPS Operating Systems Review*, 26(2):26, 1992.

[14] 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.

[15] 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.

[16] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proc. of ACM Symposium on Theory of Computing*, pages 654–663, May 1997.

[17] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: an architecture for global-scale persistent storage. *SIGPLAN Not.*, 35(11):190–201, 2000.

[18] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[19] W.-S. Li., V. S. Batra, V. Raman, W. Han, and I. Narang. Qos-based data access and placement for federated systems. In *VLDB '05: Proceedings of the 31st International Conference on Very Large Data Bases*, pages 1358–1362, Trondheim, Norway, 2005.

[20] W. Litwin and T. Schwarz. Lh*rs: a high-availability scalable distributed data structure using reed solomon codes. *ACM SIGMOD Record*, 29(2):237–248, 2000.

[21] K. Petersen, M. Spreitzer, D. Terry, and M. Theimer. Bayou: replicated database services for world-wide applications. In *Proc. of the 7th workshop on ACM SIGOPS European workshop*, Connemara, Ireland, 1996.

[22] 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 (FAST '07)*, San Jose, CA, USA, February 2007.

[23] A. Rowstron and P. Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In *Proc. of ACM Symposium on Operating Systems Principles*, Banff, Alberta, Canada, 2001.

[24] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: a highly available file system for a distributed workstation environment. *Transactions on Computers*, 39(4):447–459, 1990.

[25] M. Shapiro, P. Dickman, and D. Plainfossè. Robust, distributed references and acyclic garbage collection. In *Proc. of the Symposium on Principles of Distributed Computing*, Vancouver, Canada, August 1992.

[26] H. Stockinger, K. Stockinger, E. Schikuta, and I. Willers. Towards a cost model for distributed and replicated data stores. In *Proc. of Euromicro Workshop on Parallel and Distributed Processing*, Italy, February 2001.

[27] M. Stonebraker, R. Devine, M. Kornacker, W. Litwin, A. Pfeffer, A. Sah, and C. Staelin. An economic paradigm for query processing and data migration in mariposa. In *Proc. of Parallel and Distributed Information Systems*, Austin, TX, USA, September 1994.

[28] T. Pitoura, N. Ntarmos and P. Triantafillou. Replication, Load Balancing and Efficient Range Query Processing in DHTs. In *Proc. of Int. Conference on Extending Database Technology*, Munich, Germany, March 2006.