Differentiated Replication Strategy in Data Centers

Tung Nguyen, Anthony Cutway and Weisong Shi
{nttung,acutway,weisong}@wayne.edu

Wayne State University

Abstract. Cloud computing has attracted a great deal of attention in both academia and industry. We envision the provisioning of differentiated services as being one of the key components to the success of cloud computing. Unfortunately, this issue has not been fully addressed in the past. Realizing that different users might have different requirements regarding availability, reliability, durability, response time and so on, we conjecture that providing flexible replication mechanism is the right approach to service differentiation. In this paper, we propose *Differentiated Replication (DiR)*, which allows users to choose different replication strategies by considering both the user requirements and system capability. We implemented a system that offers four differentiated storage services with DiR. The experimental results show that this service actually provides different availabilities and execution times for different service types with the same request traces, failure traces, and workload. In addition, we also show that in comparison to the regular uniform replication, DiR can further improve resource utilization, which will in turn provide a better user experience with regards to cloud computing.

1 Introduction

Recently, cloud computing has been a hot research topic. In fact, it may shape the future of the computer industry [1–3]. With cloud computing, companies can reduce their overheads regarding the buying, installing, and maintaining computer resources. With cloud computing, they can register necessary services from the Internet, allowing them to focus on the core aspects of their business.

It is clear that many users of cloud computing services have different requirements for the service. Some require optimal performance, while others seek data redundancy and reliability. In general, users may demand different properties from services, such as availability, reliability, durability, and performance. One, several, or all of these properties may be under consideration for any given request. This possibility demands differential services. Most available cloud services today do not take this fact into account. Often times, they only provide one type of service for all of their users. For example, Windows Azure and Amazon S3 maintain a fix number of replicas (3) for data stored in it. Both Microsoft Windows Azure and Amazon S3 guarantee in their service level agreement (SLA) that the availability of customer's data is always greater than 99.9%. This may lead to poor resource utilization from the provider or an inefficient usage from the user. We also find that all of these properties relate to replication. Differentiated replication strategies can provide different availabilities, reliability, durabilities, and performances. Therefore, we propose a new strategy called *Differentiated Replication (DiR)* to address this problem.

To demonstrate the concept of DiR, we built a prototype system providing storage services capable of providing data at different rates of availabilit. Users are provided a simple interface that allows them to *store* and later *fetch* their data with their expectation. The current version of DiR provides four replication types.

This paper comes in three core parts . First, we propose the idea of differentiated services for data centers and design a set of simple but powerful APIs for high level users. Second, we propose **four** different replication strategies on the server side, enabling differentiated services in terms of data availability. Third, we implement a prototype of DiR and evaluate the four proposed replication strategies in a comprehensive manner in terms of availability using both synthetic and real failures traces. The evaluation results show that the last replication strategy, which takes both user requirements and system behavior into consideration, is capable of providing different availabilities and execution times.

The rest of the paper is organized as follows. Section 2 exhibits the design of DiR, Section 3 describes the implementation, and Section 4 presents the experiment and results. The related work and conclusion are covered in Sections 5 and 6 respectively.

2 System Design

2.1 Assumptions and Requirements

The goal of DiR is to build a read/write only (not modified) storage system that provides different types of replication services to the user with better resource utilization. We target read/write only storage system for the sake of simplicity in term of data consistency. However, this is also practical because the data in the Cloud is often very huge and should not be modified. This assumption is often made in the area of data intensive computing like HDFS of Hadoop [4].

We mainly focus on differences in: (1) replication strategies; (2) search algorithms; (3) network topology; and (4) availability. Hence, in our case *better service* may mean higher availability or faster or less communication cost. The idea for a user to request other properties such as durability, reliability, and performance is almost similar and will be introduced later. The system is heterogeneous with different hardware, software, computational ability, etc. Each member can join or leave the system or fail at any time. Given the requirement of read/write only service, the system does not need to maintain consistency between replicas. Therefore, the only two methods we need to provide are *store* and *fetch*. In addition, the failure we consider is of a fail-stop type rather than the Byzantine failure type. This means that when the machines are alive, they are supposed to have correct behavior. Finally, reasonable load balance, fault-tolerance, scalability and reliability are also important requirements of the system.

2.2 APIs

Users of our system are not terminal application users but developers of front-end applications. They are not supposed to know replication techniques in detail. The interface component is required to be simple enough so that it can easily be used in applications.

Table 1. The DiR APIs.

Function name	Description
`fetch(filename)`	retrieve a file from the DiR system
`fetch(filename,service-type)`	retrieve a file from the DiR system with specific service type
`store(filename)`	insert a file from local file system to DiR
`store(filename, service-type)`	insert a file from local file system to DiR with specified service type

Therefore, we only need to add one more parameter, called *service-type*, to the current APIs of Chord/DHash [5] to indicates which type of service the user requires. These methods merely call new methods with a default service-type, as illustrated in Table 1. The user will notice that the service-type of a *fetch* needs to match that of the *store* for a certain file.

2.3 Availability Analysis

Basically, there are two ways to provide different levels of availability: change the number of replicas, or change the location of them. Higher availability of an object can be achieved by increasing the number of its replicas or by placing its replicas onto more "available" machines. Intuitively, more replicas on more reliable nodes will produce higher availability. Even so, we cannot tell which method provides better availability in some situations. The system has to decide the availability under the resource constraints to guarantee the load balance. If we do not handle this correctly, we may introduce extra overhead to highly available nodes. The problem is formalized as follows.

Given an expected availability A of a certain object/file, and a set of nodes with their own availability, we need to find the number of replicas, and the specific nodes in which to store them. It is noteworthy that the availability of an object stored on a machine is equal to the availability of that machine, also under the fail-stop assumption.

Let M be a set of nodeIDs and corresponding availabilities of N nodes.

$$M = \{(n_i, a_i) | n_i \text{ is the ID of node i and } 1 \leq i \leq N\}$$

Let

$$\sigma = \left\{ \{(n_{x_l}, a_{x_l})\}_{l=\overline{1,k}} \left| \begin{array}{l} (n_{x_l}, a_{x_l}) \in M, \\ n_{x_i} \neq n_{x_j}, 1 \leq i, j \leq k, \\ 1 \leq x_i, x_j, k \leq N \end{array} \right. \right\}$$

The solution of the problem is in σ set.
Assuming $\{(n_{y_1}, a_{y_1}), (n_{y_2}, a_{y_2}), \ldots, (n_{y_l}, a_{y_l})\}$ is one specific solution, the following approximation should be satisfied

$$\begin{aligned} A &\approx 1 - (1 - a_{y_1})(1 - a_{y_2}) \ldots (1 - a_{y_l}) \\ &= f(a_{y_1}, a_{y_2}, \cdots, a_{y_l}). \end{aligned} \quad (1)$$

Note that in Equation (1) while A represents the user's expectation, $\{a_{y_1}, a_{y_2}, \cdots, a_{y_l}\}$ represents the availability of the system. With a certain value of A, we may have several solutions.

One way to completely solve this is: For every member m_i of σ, compute $A' = f(m_i)$, if $A' \approx A$ then m_i is a solution. Unfortunately, this method is $O(2^N)$, in which N is the number of nodes.

The second method is to calculate the average of all the availabilities

$$\overline{a} = \sum_{i=1}^{N} a_i$$

and use the formular in [6]:

$$l = \frac{log(1 - A)}{log(1 - \overline{a})} \tag{2}$$

to derive the number of replicas l. This approach may create a resource utilization problem in the next step of choosing proper nodes to store replicas.

The third method is shown in the following algorithm

```
Sort $M$ in decending order of availability
$F \Leftarrow (1 - M[0].a)$
$i \Leftarrow 1$
while $(i < N) and (A' < A)$ do
    $F \Leftarrow F * (1 - M[i].a)$
    $A' \Leftarrow 1 - F$
    $inc(i)$
end while
```

This method may cause an overload in the high availability nodes.

Finally, since this problem is of a constraint programming type, another regular way to solve it is to use an existing C(L)P solver.

As a result, no matter what method we use, from the system design point of view, the system is required to have a monitor server to provide the availabilities of all nodes in the system. This leads to the need for *OPERA*.

2.4 *OPERA*

OPERA stands for **OPE**n **R**eputation **A**rchitecture, which is a general framework to compute the reputation of nodes in the system. *OPERA* allows users to define how to calculate reputation, and it returns the reputation of nodes based on that definition. It employs a traditional master-slave model in its communication, since we need to obtain the global reputation of the system. *OPERA* clients communicate to each other in replying to the rate request from the server. Design and implementation of *OPERA* are not detailed here due to page limitation. Basically, OPERA employs Ganglia (a monitoring tool) to collect information about nodes in the system and calculates reputation for each node based on this information.

2.5 Utilization Analysis

We argued that DiR also provides better resource utilization. This is rather obvious and straightforward. Let's define

$$\text{the utilization of a system} = \frac{Res_{real}}{Res_{need}}$$

in which, Res_{real} are the resources that really used to provide the services and Res_{need} are the resources that can satisfy user needs.

The following analysis compares the resource utilization of DiR and that of uniform replication, which is a very widely used technique today. We assume both systems have n requests $r_1, r_2, \cdots, r_n$ and c_i is the correspondent number of replicas to satisfy r_i. The total number of replicas of DiR (idealy) and uniform method is $\sum_{i=1}^{n} c_i$ and $\sum_{i=1}^{n} Max\{c_i|0 \leq i \leq n\}$ respectively. Note that, in the uniform repication system, we need to choose l large enough to satisfy the highest quality requests. For example, with $n = 4, c_1 = 2, c_2 = 3, c_3 = 2, c_4 = 4$, the resource utilization of ideal DiR and uniform system is 1 and $\frac{4\times4}{2+3+2+4} = 1.45$. As a result, this analysis proved that DiR used resources more efficiently. From another aspect, with the same resource, DiR (better utilized system) can satisfy more requests as well.

2.6 DiR System Architecture

The overall system architecture is shown in Figure 1. The user uses the DiR Interface to ask for service. Depending on the request, the DiR interface decides which replication strategy to use. There are, in total, four replication strategies available (represented by four blocks in the figure) that can offer all required differences. In fact, there are many other options to choose to construct a strategy. For example, we can choose random walk search algorithm [7], CAN [8] or Pastry [9] to build a new type of service. Such openness is expressed by the lowest block with "three dots" in the figure. Finally, the rightmost circle with small squares inside represents the physical underlying network connecting the machines of the system.

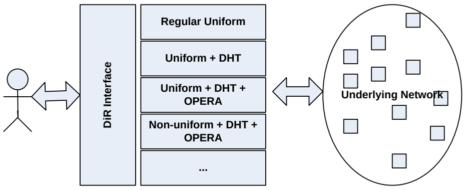

Fig. 1. The DiR system architecture.

Our system can also support the differences in the durability/reliability of objects stored in it by modifying the policy being used in the monitoring server, *OPERA*, to define how to calculate them. The nodes that are more durable/reliable have higher reputation scores. By doing this, the *OPERA* server returns the durability/reliability of all nodes in the system. The remaining problem is how to calculate these values.

Table 2. The summary of available service types

Service type	Description
S_d	Regular uniform replication, unstructured network
S_d	Uniform replication, DHT, ring-based
S_{d+o}	Uniform replication, DHT, ring-based, OPERA
S_{d+o+a}	Non-uniform replication, DHT, ring-based, OPERA

3 Implementation

We implemented our system by extending Chord/DHash [5]. The architecture of an individual node (peer) derived from our previous design is shown in Figure 2. The rightmost block is the OPERA client. This block is in charge of rating other nodes and responding to the request from them as well as from the OPERA server. The highest level in Figure 2 is the DiR interface that offers the APIs to the user and chooses an appropriate handler. The two leftmost blocks (File Transfer and breadth-first search) correspond to the “Regular Uniform” handler of Figure 1. “File Transfer” is used to receive files sent from other nodes. ”Breadth First Search” is used to find a replica. The last three middle blocks (DiR Manager, DHash and Chord) correspond to the final three handlers in Figure 1. The lowest block, Chord, is a replica lookup service. The upper block that uses the Chord lookup service is DHash, a block store service. This layer is responsible for storing/retrieving blocks of data into/out of storage devices. The DiR Manager is located on top of the block store layer (DHash). It is used to provide the `store/fetch` file functions to the DiR interface, communicate to OPERA and calculate the appropriate number of replicas as well as their locations.

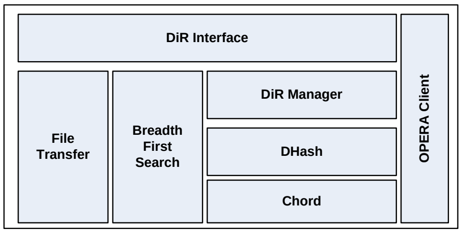

Fig. 2. An augmented node in DiR implementation.

We have implemented a prototype of a DiR storage system and OPERA using C++ on Linux. The DiR prototype has implemented all parts of the previous design. It offers several types of differences: in the network topology (unstructured or structured), communication model (message and aRPC), search algorithm, replica location and expected availability. These differences are implied in the following four service types (corresponding to the four handlers in the Section 2.6) and are summarized in Table 2.

Type S_r indicates that a user wants to use a normal uniform replication with a traditional search algorithm. Nodes communicate with each other by sending and receiving messages. Although this strategy is popular, we built it from scratch since we could not find any open source implementation available online. To handle this type, each node has two servers. The first one, (called searchserver), waits to receive a file request and looks for the file in the system. This server is the "Breadth First Search" block in Figure 2. To make the search process workable, we apply the TTL (Time To Live) technique to each request. The second server, (called filetransferserver), is used to receive actual files sent from searchservers. It is worth noting that since we use failure traces in the experiment, these servers have to be implemented to tolerate failure at anytime. Therefore, we also apply a timeout technique in the communication.

Type S_d represents the uniform replication, using Chord as a DHT lookup algorithm. In fact, this type is the original version of Chord/DHash.

Type S_{d+o} utilizes the same lookup algorithm as S_r but with a different replica location. With this service type, DiR first calls Chord to get the successor nodes of the hash value of the filename and contacts OPERA to ask for their availabilities. Based on this information and the predefined number of replicas, DiR chooses nodes with the highest availability to host the replicas of that file. The thing worth being noticed here is that in this prototype, these first three types have **the same number of replicas**.

Type S_{d+o+u} means the user only cares about the availability, not the number of replicas. This version of DiR **fixes the target availability to 3 "nines"** as in Amazon S3 storage service, but this can easily be changed to the desired values. DiR calculates the necessary replicas using equation (2) with the average availability of the successor list of the file ID. In the step of choosing host machines to store the replica, we simply choose the sets randomly, recalculate the availability accordingly of each set, and choose the one that is close to the expected availability. Although this way may not provide the best solution, it is faster and helps to balance the load.

4 Experiment and Results

To evaluate our system, we deployed DiR onto a cluster of 21 nodes. The first 20 nodes had DiR installed. The 21 node was dedicated to the OPERA server with a guarantee not to fail during the experiment.

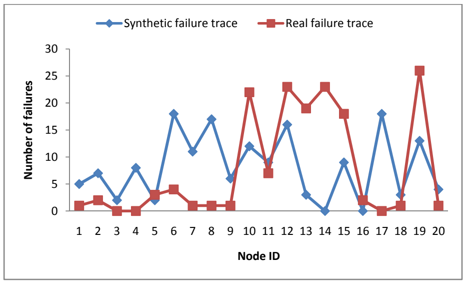

Fig. 3. Number of failures.

To prepare for the experiment, first, we need to have a files generator and distributor to create and distribute files randomly to 20 nodes. The total number of files in the synthetic trace is 400, and that of the DZero trace [10] is 25,951 files.

Second, for request and failure traces, we used both synthetic traces and modified real traces. Figure 3 shows the detailed number of failures in both synthetic and real failure trace. The total failure duration time of each node is displayed in Figure 4. The simulation time for the synthetic traces is 50 minutes. The real failure trace is from the availability information of the first 20 nodes of Microsoft PCs trace [11] measured during 35 days since July 6, 1999; and the real request trace is from a physical application of the DZero experiment [10] on April 2004. Since it was not practical to conduct the experiment for an entire month, both real failure and request traces are scaled down to one day only. In scaling down the request trace, we encountered the problem of congestion. This is because the request trace was forced to request too many files at the same time. As a result, we modified the trace so that the time to ask for the file is slightly different if they are the same in the original trace. The availability results returning from the OPERA server are shown in Figure 5.

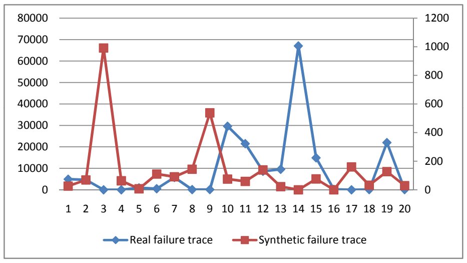

Fig. 4. Failure duration time in the failure trace.

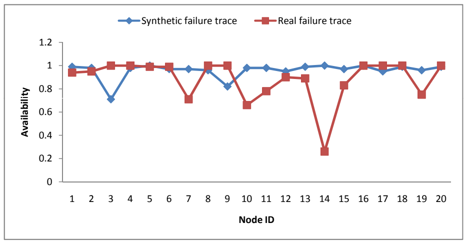

Fig. 5. The availability of the system.

In this section, we measure two main metrics: *execution time* and *availability* of different service types.

Table 3. The availability of DiR under synthetic and real traces.

Service-type	$S_r(2)$	$S_r(3)$	S_d	S_{d+o}	S_{d+o+u}
Availability (synthetic trace)	44.62%	86.40%	61.62%	70.06%	98.13%
Availability (real trace)	28.02%	29.32%	82.17%	89.79%	99.95%

Table 3 shows the availability of the system using the synthetic traces and the modified real traces. From the synthetic results, we can see significant improvement in the availability of the S_r with different configurations of the neighbor list (two or three neighbors). In addition, we also found that the availability of S_{d+o} is better than that of S_d. This means we can improve the availability of Chord/DHash with the aid of OPERA. The table also shows that we cannot tell which type is "better" generally. One can argue that $S_r(3)$ (with 3 neighbors) is the best. However, this statement is true for the availability only. It is easy to see that, the $S_r(3)$ strategy costs more resources (in term of link number and bandwidth) than the others. From the real results, the availability of S_{d+o} is also better than that of S_d. The availabilities of S_r are poor because the real request trace requests several hundred files at the same time; and together with the failure trace, it crashed some of our search servers and hence, produced those poor availabilities.

Another experiment was about S_{d+o+u} alone. Figure 6 shows the availability of 10 different files of random size. The horizontal line in the figure represents the expected availability that was set to three nines by default. The result was measured in 50 minutes.

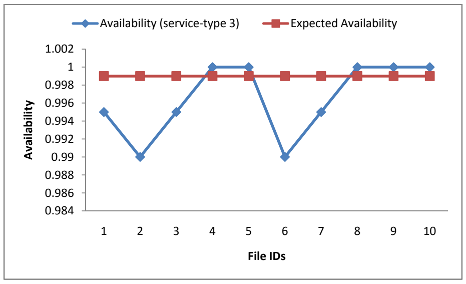

Fig. 6. Availability of files using S_{d+o+u}.

To measure the performance of DiR, we created files with various sizes, inserted them into the system and then retrieved them. We only measured the response time of the successful requests. Assuming that there was no failure, we got the results shown in Figure 7.

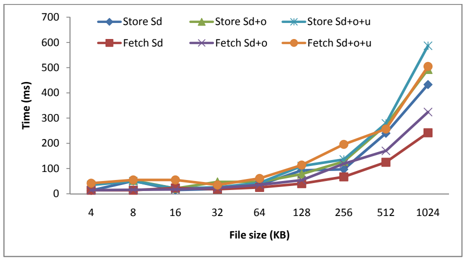

Fig. 7. DiR performance of S_d, S_{d+o} and S_{d+o+u}.

5 Related Work

Differential service is one of the key aspects of DiR, encompassing flexible availability, reliability, durability, file placement, and search methods. Many studies are related to and have led to the culmination of DiR. Beside Chord/DHash [5], perhaps the closest work to ours is the Total Recall [6]. Total Recall file system of Bhagwan *et al.* also provides an option to choose the availability of objects. However, their users are in fact system administrators. They aimed at relieving administrators' burden by maintaining the degree of replicas automatically. Another paper of Zhong *et al.* [12], which also employed the non-uniform replication, considers the optimal number of replicas of an object for high availability to be directly proportional to the object's popularity. However, their approach does not consider the different type of services from users. Also related to replication strategy, Cohen [13] finds the optimal replication strategy, in term of search size, lies between a uniform and a proportional strategy.

Different placement algorithms were introduced in [14–17]. To increase availability, Giwon's work in [18] is concerned with dynamically replicating objects, and Acunam's group uses a fixed number of replicas depending on peer availability [19]. In regard to durability, Wang *et al.* focuses on the durability of data through replication [20], while Chun's research improves the durability of large amounts of data using a replication algorithm [21]. Besides the breadth first search and the Chord mentioned in the previous sections, [7] presented a search algorithm using multiple random walks to improve performance over the Gnutella like flooding search method.

6 Conclusions and Future Work

Diversity in the requirement of services will soon be an important feature and requirement in cloud computing. In this work, we took the first step to address this problem. Focusing on the replication technique, we proposed the concept of differentiated replication that can offer different types of services, and developed a prototype storage system focusing on the availability.

Future work in this project will involve improving OPERA so it will be capable of monitoring other metrics, such as performance, durability, and so on. In addition, we will investigate the techniques to maintaining multiple service types in the dynamic environments, e.g., large-scale data centers [1], which have many different applications simultaneously.

7 Acknowledgments

We would like to thank the anonymous reviewers for their comments and Cole Brown at Rochester for his editing changes on this manuscript.

References

1. Bryant, R.E.: Data-intensive supercomputing: The case for disc. Technical Report CMU-CS-07-128, School of Computer Science, Carnegie Mellon University (May 2007)
2. Armbrust, M., Fox, A., Griffith, R., Joseph, A., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., et al.: Above the clouds: A berkeley view of cloud computing. EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-28 (2009)
3. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., Zaharia, M.: A view of cloud computing. Commun. ACM **53**(4) (2010) 50–58
4. : http://wiki.apache.org/hadoop/
5. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. In: ACM SIGCOMM'2001. (2001)
6. Bhagwan, R., Tati, K., Cheng, Y.C., Savage, S., Voelker, G.M.: Total recall: System support for automated availability management. In: Proc. of NSDI'04, Berkeley, CA, USA, USENIX Association (2004) 25–25
7. Lv, Q., Cao, P., Cohen, E., Li, K., Shenker, S.: Search and replication in unstructured peer-to-peer networks. In: ICS '02: Proceedings of the 16th international conference on Supercomputing, New York, NY, USA, ACM (2002) 84–95
8. Ratnasamy, S., Francis, P., Handley, M., Karp, R., Schenker, S.: A scalable content addressable network. In: Proc. of ACM SIGCOMM'01. (2001)
9. Rowstron, A., Druschel, P.: Pastry: Scalable, distributed object location and routing for large scale peer-to-peer systems. In: IFIP/ACM Middleware 2001. (2001)
10. Doraimani, S.: Filecules: A New Granularity for Resource Management in Grids. Master thesis in Computer Science, University of South Florida (2007)
11. Bolosky, W.J., Douceur, J.R., Ely, D., Theimer, M.: Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. In: Proc. SIGMETRICS. (2000)
12. Zhong, M., Shen, K., Seiferas, J.: Replication degree customization for high availability. SIGOPS Oper. Syst. Rev. **42**(4) (2008) 55–68
13. Cohen, E., Shenker, S.: Replication strategies in unstructured peer-to-peer networks. SIGCOMM Comput. Commun. Rev. **32**(4) (2002) 177–190
14. Chervenak, A.L., Schuler, R.: A data placement service for petascale applications. In: PDSW '07: Proceedings of the 2nd international workshop on Petascale data storage, New York, NY, USA, ACM (2007) 63–68
15. Chervenak, A.L., Schuler, R., Ripeanu, M., Amer, M.A., Bharathi, S., Foster, I., Iamnitchi, A., Kesselman, C.: The globus replica location service: Design and experience. IEEE Transactions on Parallel and Distributed Systems **99**(1) (2008)
16. Douceur, J., Wattenhofer, R.: Optimizing file availability in a secure serverless distributed file system. Reliable Distributed Systems, 2001. Proceedings. 20th IEEE Symposium on (2001) 4–13

17. Lian, Q., Chen, W., Zhang, Z.: On the impact of replica placement to the reliability of distributed brick storage systems. In: ICDCS '05: Proceedings of the 25th IEEE International Conference on Distributed Computing Systems, Washington, DC, USA, IEEE Computer Society (2005) 187–196
18. On, G., Schmitt, J., Steinmetz, R.: The effectiveness of realistic replication strategies on quality of availability for peer-to-peer systems. In: P2P '03: Proceedings of the 3rd International Conference on Peer-to-Peer Computing, Washington, DC, USA, IEEE Computer Society (2003) 57
19. FM, C.A., Martin, R., Nguyen, T.: Autonomous replication for high availability in unstructured p2p systems. Reliable Distributed Systems, 2003. Proceedings. 22nd International Symposium on (Oct. 2003) 99–108
20. Wang, A.I.A., Reiher, P., Kuenning, G.: Introducing permuted states for analyzing conflict rates in optimistic replication. In: SIGMETRICS '05: Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems, New York, NY, USA, ACM (2005) 376–377
21. Chun, B.G., Dabek, F., Haeberlen, A., Sit, E., Weatherspoon, H., Kaashoek, M.F., Kubiatowicz, J., Morris, R.: Efficient replica maintenance for distributed storage systems. In: NSDI'06: Proceedings of the 3rd conference on Networked Systems Design & Implementation, Berkeley, CA, USA, USENIX Association (2006) 4–4