Using Cooperative Caching and Replication to Improve Reliability

Salma Rodriguez Jorge Cabrera Ming Zhao
Florida International University
{srodr063, jcabr020}@fiu.edu
ming@fiu.edu

Abstract

The method of using client-side SSD caches on distributed storage systems has been limited mostly to write-through configurations. Even though write-back SSD caching provides better I/O performance by reducing the overhead write requests from a faster storage device instead of the slower storage device, the potential risks of data loss coming from client failure or cache device failure can deter users from adopting these configurations in their systems. This project offers a strategy for increasing reliability guarantees of write-back caches by implementing cross-client cooperative caching and replication policies. The results of this research could potentially allow for a wider adoption of write-back caches in storage systems.

1 Introduction

Client-side block-level caching has been shown to relieve storage area network systems from load balancing and scalability issues [3][9]. By using fast storage devices such as flash-based SSDs as a local caching medium, clients can support better I/O performance compared to using only the storage system network. Despite the I/O performance benefits of a storage caching layer, reliability issues can arise if dirty data is not persisted to the storage back-end frequently. On the one hand, write-back policies offer great performance benefits as they may take advantage of requests with temporal locality and reduce the load on the server. However, if the flushing of dirty data is postponed for too long, a client failure may result in critical data loss. Other policies such as write-through offer better reliability guarantees but discard any performance gains obtained from buffering writes. Our solution consists of a block-level caching system that implements cross-client cooperative caching and replication policies, to compromise reliability and I/O performance gains.

The remainder of this paper is organized as follows. In section 2 we provide background information on replication and cooperative caching. Section 3 describes our proposed approach for establishing reliability using cooperative caching and replication. Finally, section 4 contains a brief schedule of the tasks that will take place to implement and test our cooperative caching and replication code.

2 Background

Replication techniques for distributed systems have been explored on many papers. Optimistic replication has been devised in order to increase availability, while allowing concurrent access to data [5]. Although previous research has focused on improving availability on distributed [2] and grid computing systems[4], reliability for distributed caching systems has had little improvement.

Pessimistic replication is a synchronous replication technique that guarantees a high degree of consistency and reliability by blocking access to data until all destinations have received a consistent copy of the data. However, pessimistic replication increases the variability in access time, since network behavior and hardware failure are highly unpredictable. As a result, latency spikes are expected to occur more frequently. In the case of DM Cache, using pessimistic replication would result in higher latency when there are more clients cooperating as replication targets. The scalability problem with pessimistic replication is shown in [8], as well as [7].

The work of Saito and Shapiro on optimistic replication [5] shows how to achieve eventual consistency on distributed systems while making the data highly available across different locations on a geographic network. However, there is no warranty for reliability, since the data is replicated asynchronously, with the optimistic assumption that the target will receive the data.

Our focus for this paper is not to achieve high availability, but we will use some of the ideas behind opti-

mistic replication in order to maintain some availability. In all cases, it is not possible to make the system both highly available and highly reliable. To have good reliability there must be a reasonable number of consistent copies of the data after every update. Reliability is directly proportional to the number of consistent replicas in a distributed caching system, where each client has access to its own data and the data of no other client.

We employ a technique that is similar to that used in [1], where a fixed number of clients holding the replica are required to respond before proceeding with subsequent operations. The number can be dynamically changed according to frequency of access, so that data that is written to more frequently mandates a higher number of responses for each replication operation. Other approaches, including optimistic replication, can be statistically compared to this approach, in order to determine which makes the system more reliable, and which is best under which circumstances.

Consistent hashing has been explored in many works of research. By using a key-value approach, the hash function can be designed so that there is an upper bound to the number of keys that each node is responsible for. An advantage to using a circular hashing mechanism is that nodes can join or exit the network dynamically, and affect only the nearest successor [6]. If time allows it, we will explore consistent hashing and how it can improve reliability by giving each client a fair chance to replicate its content.

3 Approach

In our initial prototype, replicas are distributed synchronously to ensure reliability. Our model comprises a block-level, distributed caching system where unwritten, dirty data is replicated across the caches of other clients in the system. We use iSCSI as our cooperative caching protocol, to make clients in the network aware of the cache capacity of their peers and utilize any available free storage to propagate replicas of dirty data. In the future, we would like to consider the tradeoff between free space and the degree of replication in our prototype, so that each client has a fair chance to replicate their data whenever there is an update.

3.1 Design

In our design, we assign a cache partition for holding replicas from other clients. Device Mapper Cache then uses the partition as a replica cache, where replicas are sent to it by a peer client through TCP/IP and given a time stamp to ensure consistency, since consistency cannot be guaranteed, especially when not all of the replica clients respond to a replication request. We define a *captured page* to be a dirty page belonging to another client. The *degree of replication* is equivalent to the number of captured pages in a client. A *replica target* is a client with enough free space in its local cache to satisfy a replication request. Clients replicate data proactively in the background. During propagation, write operations are queued until each replica reaches its replica target.

A timeout is specified in order to avoid blocking the writes for too long. The timeout, which indicates a target failure, is based on the network latency and can be adjusted depending on network traffic. We also use a "heartbeat" daemon to immediately flush the dirty data when the client holding the original data fails. The daemon works as follows: the replicator client will synchronize with target clients so that at fixed time intervals, the target will read a message passed through TCP indicating that the replicator client is running and that its cache is available. If either of these conditions fail, the data is immediately flushed back by the replica client, chosen from a list of clients with the most up-to-date copy in order to ensure consistency.

In our evaluation of reliability, we consider different approaches for handling failure, including power failure, node failure, disk failure or failure due to network partition. We analyze various design tradeoffs for failure handling, exploring whether or not writing back to a RAID on the storage server makes the system more reliable than having multiple attempts to replicate the data.

If time allows, our design will include a cache partitioning algorithm that considers the tradeoff between free space and the *degree of replication* in each client in order to perform fair replication across clients, and ensure system-wide reliability. The benefit for choosing a replication candidate with enough free space is as follows:

$$benefit = \frac{f_i}{r_i} \tag{1}$$

where f_i is the free space in node i, and r_i is the degree of replication in node i. Dynamic network joins and exits will not be considered for this paper. Therefore, we will use a circle of peer clients for our initial prototype. We will also partition the cache of each client so that a small percentage is allocated to the replicas of its peers, leaving the implementation of an intelligent cache partitioning algorithm for future work.

Figure 1 shows the general structure of our prototype. In our framework a node can serve any of the following roles: replicator, replica target, or storage server. The sections below explain each of these roles.

3.1.1 Storage Server

A storage server node in our system is the machine that holds the primary storage medium used by the storage

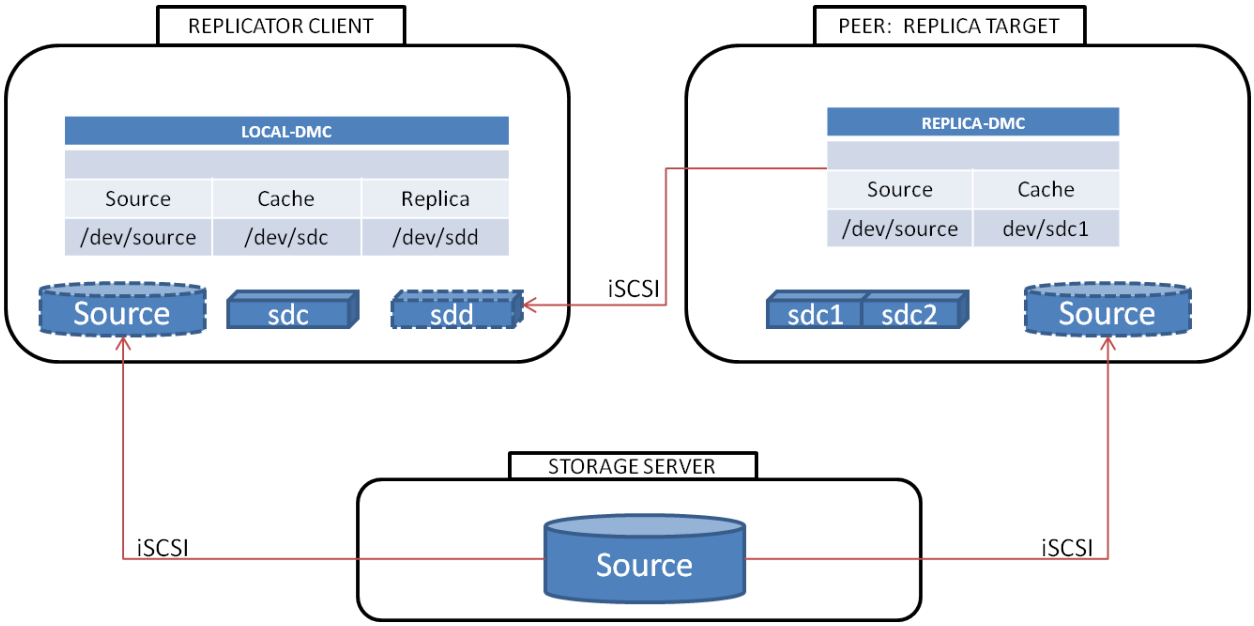

Figure 1: The general structure of our prototype. A node can serve any of the following roles: replicator, replica target, or storage server.

area network. This storage is made available to to other client machines by way of an IP-based protocol, such as iSCSI. For our prototype the storage server node provides a source storage device to both the replicator client and the replica peer target by using the iSCSI protocol. The replicator and replica target see a logical version of the source device and can operate on it just as it if were physically attached.

3.1.2 Replica Target

A replica target node is a client that shares its local SSD cache device with other clients. This client has two special functionalities. The first functionality is to handle the copies of dirty data that are sent to it by other clients who wish to use it as a recipient for their replicas. The second functionality is to serve as a potential source of data recovery in case the client with the original data fails. The mechanism used to accomplish these two functionalities is done by implementing a special version of the DM Cache module which has been modified to hold replicas from other clients, and to flush the versions of dirty data in a recovery scenario. The reason we use a special version of DM Cache in the peer client is because it provides us with a logical device which we can present to the replicator client as a cache device specifically for replicas. The replicator client can see this device as a local device by using a protocol like iSCSI. More details of the implemention of this DM Cache module can be seen in the implementation section of this paper.

3.1.3 Replicator

A replicator node is a client that wishes to replicate its dirty writes onto other peer clients. This client runs a special version of DM Cache which allows it to specify a replica cache device, a local cache device, and source device when loading. The current version that we implemented for our prototype only allows it to specify a single replica cache device, however, this feature can be simply expanded so that when loading our version of DM-Cache on a client machine, it will have the chance to specify a circle of peer cache devices to which it can propagate replicas. A *circle* would just be a group of clients which have agreed to let another peer client use its free cache space.

3.1.4 Communication Protocol

In order to propagate dirty writes to other peer clients a replicator client performs write operations in a replica cache device, in addition to the local cache device. This replica cache device is the virtual block device made available through iSCSI and it is generated by the replica version of DM Cache running in the peer client.

The replicator client and the replica target only need to set up an iSCSI initiator module and an iSCSI target, respectively. This mechanism allows to transfer data from one client to another without having to modify much of the DM Cache source code and makes it simple to propagate replicas of dirty data to other devices, whether physical or logical. When a propagation request is activated,

a client will send out a request to each client in its circle and if not enough clients respond with a confirmation, it can be assumed that there was not enough free space, or that there was a failure in some of the clients.

The reason for replicating in a static fashion rather than dynamic is because we first want to focus on creating a communication channel that is stable and that has acceptable performance in comparison to the baseline case. If time allows it, a best-case scenario implementation will result in a system allowing a client to join a network of cooperative clients without having to reload DM-Cache with a new circle of clients, as well as using the intelligent paritioning algorithm discussed earlier.

3.2 Implementation

The implementation of our prototype will use DM Cache, which is an existing block-level caching solution. We will modify it to incorporate our cooperative caching and replication mechanisms. The current implementation of DM-Cache exists as a Linux kernel module. It intercepts block I/O requests directed towards the main storage device, and redirects these towards a cache device. Information about redirected requests, such as source device to cache device block mappings, block status flags, and other related meta-data is kept in a special data structure. Our prototype will use DM-Cache, its block request mapping and its redirection mechanism as the foundation of the block-level cache management mechanism.

As part of this project, iSCSI is used as the storage communications protocol for DM Cache. In the current setup of DM Cache, where clients work in a non-cooperative manner, the storage server functions as the iSCSI target, and the client machine functions as the iSCSI initiator. In other words, the client machine would only access the storage resources allocated to it in the server. In order to transfer blocks of data from client to client, the new system will require that all clients have access to the cache of their peers. Having communication established among peer clients, each client can then initiate replication processes or act as target for replicas.

The sections below explain in further detail the implementation of the replication, communication, and recovery mechanisms that were implemented as part of this project.

3.2.1 Replication

Our prototype depends on two special versions of DM Cache in order to perform the replication and storage of dirty data. The first version of DM Cache that we worked on was the one that replicated data for dirty writes. We will refer to this version as "propagator-dmc".

The original version of DM Cache allows one to specify a source device and local cache device. In our modified version one can specify a third device which we will refer to as the replica cache device. This device is just the logical representation of the device which is made available using iSCSI, and it's produced by the replica version of DM Cache running in the peer client. The advantage of using this method to propagate data for dirty writes is that there needs to be little modification on the DM Cache source code. This is because DM Cache uses a special function from the Device-Mapper function library. The function is dm_io(). This function allows DM Cache to perform read or write operations synchronously or asynchronously. One special function which we took advantage of is that this function these services allows us to take a write bio request and redirect it to more than one block device. All we need to specify for each destination is the block device, the address, and the size of the request. The synchronous function will block until a request has been dispatched to both devices. The asynchronous version allows us to specify a callback function to handle the completion of both devices, so it does not block.

Besides the block device, the difference between data going to the local cache and the replica cache is the address to which they're directed. For local caches, the "propagator-dmc" version of DM Cache directs a write request to a section in the local cache device that is specified by the current cache management policy (the default configuration we used is based on a set-associative cache). As for the replica cache device, the address of the request is the same as the original bio request. This is done so that when this request is received by the replica client it stores a mapping for the original address, the one pointing to the source device. In the case of recoveries, the dirty data is flushed to the source device using this address.

The "propagator-dmc" version handles the two cases in which a write request in DM Cache results in dirty writes. The first case is when a request results in a cache hit. The second one is when a request results in a cache write miss. In both of these cases, "propagator-dmc" replicates the dirty data to both the local cache device and the replica cache device. In addition, to handling the cases where dirty data is produced, "propagator-dmc" must also handle cases when dirty data is flushed to the source device and is made clean (valid). Luckily, DM Cache handles these cases by using a function called write_back(). This function simply copies dirty blocks to the corresponding address in the source device. For our project we modified it so that it would also send a modified bio to the replica cache device when a write-back operation occurred. This modified bio contained a special flag, signalling that the data for this block was no

longer dirty.

3.2.2 Replica Management

The second type of DM Cache that we worked on this project was the version that is in charge of running on the peer client that stores the replicas. We will refer to this version as the "replica-dmc" version. The main function of this version of DM Cache is in managing the copies of the dirty data from other clients. For our project, we modified DM Cache so as to produce a version that can handle the replicas for only one client per instance. In order to handle the replicas from other clients another instance of "replica-dmc" would have to be loaded. When loading this version of DM Cache, the parameters we pass are the source device of the replicator client, and the partition of the peer's SSD cache device. Figure **??** illustrates how this works.

The first characteristic of this version of DM Cache is that it is only used to hold dirty data from another client, and it's not designed to be used for reading or for mounting. In order to make it do this, the cache_map function of DM Cache is modified to ignore any read bios, so that only mappings for dirty writes are stored. Currently we used DM Cache's internal structure to store the mapping between the original bio's source address and the replica cache address.

The second characteristic of the "replica-dmc" version of DM Cache is that it should discard any data for blocks that are no longer dirty on the replicator client-side. This is done by checking for hints on specially-marked bios coming from the replicator client. Basically, any bio that is received in the cache_map function is checked to see if a special flag has been set by the replicator client. Any bio that has this flag set is sent to a function that changes the mapping for that block from a dirty state to a valid state, that way this mapping is ignored for recovery purposes or is replaced if space for a new mapping is needed.

3.2.3 Recovery: Heartbeat Daemon

The heartbeat daemon has a client component that runs on "propagator-dmc". The client component is a kernel thread that generates a message at fixed time intervals. The message is sent only if the replicator has not failed, and if the local cache of the replicator is available. The "replica-dmc" uses a web server that expects a *pulse* from the "propagator-dmc" after a fixed amount of time plus a reasonable amount of network delay. If the replica client does not recieve a message from the replicator, then it can be assumed that the replicator, which originally contained the data, failed unexpectedly.

The "replica-dmc" includes a mechanism for consulting with other neighboring caches, to see which one contains the most up-to-date copy of the data, in order to maintain consistency, by writing back the latest replica, in the advent that not all replica clients received the latest copy on the update that occured just before the failure. This mechanism includes metadata about each block, with a time stamp that is written on each commit and hashed by sector number.

When a failure occurs, after bootup, the kernel module will be loaded, along with the circle of clients. The module will warm the cache from the data stored in the storage server disk, since the data would have been flushed immediately after the failure from a replica client with the most up-to-date copy of the data.

4 Schedule

Current Progress

We have designed ways to replicate data and cooperate among clients. Implementation has begun, and we are still working on our prototype. Our current implementation includes ways to submit block I/Os to more than one device, and a working peer to peer communication protocol using TCP.

Jorge is working on cache partitioning and replication. He is figuring out a way to submit block I/Os as a bio request to more than one device. He will also implement the partitioning algorithm that will be used to store replicas for peer caches.

Salma is working on serialization of block I/Os which will be sent through TCP sockets across the network, and reconstructed by the receiving clients. Ways to deserialized block I/O data and submit these to the block layer queue for writing to the block device are still being implemented.

November 15 - 21

Jorge will continue working on the replication policies. Salma will continue working on cooperative caching. Jorge needs to find a way to partition the cache, so that a portion of it is reserved for peer clients. He also needs to explore ways to pin the pages of peer caches, so that the data is not accidentally erased while unwritten dirty data is still hanging around, waiting to be written back to the storage server. Salma needs to find a way to serialize block I/O data structures, write these to a socket file discriptor, with the data being read, and deserialized from the target client. Salma will implement a way to construct the block I/O request and submit for writing to the device. The mechanism will include an acknowledgement with the sector number where the block of data was stored.

November 21 - 28

A table with client metadata still needs to be implemented, so that information about replicas is kept in volatile memory and also reflected on persistent storage periodically. The implementation of replication will also include a way to mark the data as invalid in the replica targets when replicas have been written back to disk and are no longer needed. Salma will work on creating and persisting the metadata, and Jorge will implement a mechanism for invalidating replicas after every writeback. Both Jorge and Salma will be meeting with the professor and classmates for feedback and suggestions as the deadline approaches.

November 28 - 5

Use feedback and suggestions to finish whatever is missing before December 1st, and from December 1st to the 5th, both Jorge and Salma will be preparing the presentation and the classroom demonstration. Both Jorge and Salma will add figures to their slides and add bibliographical refrences to improve the quality over their first presentation.

December 5 - 12

Add whatever is missing to the paper from December 5 to 9. We realize that we need figures in our paper. These will be added along the way. Both Jorge and Salma will polish the paper before turning it in on December 12.

5 Evaluation

We use two approaches for our evaluation. The first is a practical approach, where we use one of the failure types introduced in section 3 to evaluate the design for reliability. The failure type we use is induced power failure, where power failure will purposely be induced simply with one or two replicas in the caching system. We also provide performance metrics for various replication approaches.

The second approach uses statistical methods. Reliability is quantified as mean time to failure (MTTF). The time-dependent mean time between failure (MTBF) is given by the equation:

$$\theta = \frac{T}{N} \tag{2}$$

where MTBF converges to MTTF as the number of failing devices approach the total number of devices in the system. In this equation, T is the total time, aggregated for all the nodes, so that if there are n nodes in the system and each performs I/O-bound activities on its locally attached SSD for time t, then the total time, T is

$$n \times t$$

Our hypothesis is that with a higher number of replicas in the system, the number of failing caches in proportion to T would be lower, and this is what we wanted to prove through statistical methods. Due to limitations, we were unable to proceed with a full reliability study. Therefore, our results are incomplete.

Permitted that we have the hardware and time, our future work will make use of our statistical parameter

$$\lambda = \frac{1}{\theta} \tag{3}$$

plotted against time to graphically determine the reliability curve for real workloads. This curve is given by the following equation:

$$f(t) = \lambda e^{-\lambda t} \tag{4}$$

MTTF can be accurately determined by integrating (4) over time as follows:

$$\frac{\int_0^\infty t f(t) dt}{\int_0^\infty f(t) d(t)} \tag{5}$$

This, of course, then reduces to:

$$\int_0^\infty t f(t) dt \tag{6}$$

since $f(t)$ approaches 1 as t approaches ∞. Finally, we can integrate $f(t)$ to generate a cummulative distribution function (CDF), as follows:

$$F(t) = \int_{t_0}^{t_1} \lambda e^{-\lambda t} dt \tag{7}$$

but when t_0 is zero, this integral simply reduces to:

$$F(t) = 1 - e^{-\lambda t_1} \tag{8}$$

which we will use to calculate the probability of failure for $t < t_1$. The probability of failure for $t_1 > 1$ is accurately given by

$$R(t) = 1 - F(t) \tag{9}$$

As part of the evaluation of our prototype, we measure the impact that the replication mechanism has on system. For our baseline configuration we test the performance of the original version of dm-cache that uses a local

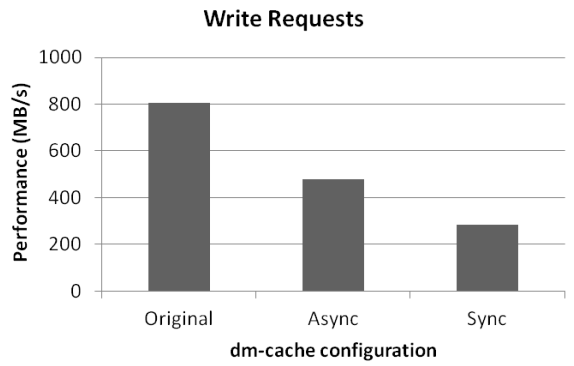

Figure 2: Performance for write operations on three different dm-cache configurations

SSD cache only. We compare this to overhead produced by two configurations of our versions of replicator-dmc. The first of these configurations uses the synchrounous version of the dm_io() function. We decide to test this function since it performs writes to both the local and peer cache devices in a synchrounous manner, and thus it's more reliable, although at a performance cost. The other configuration that we test is the version of dm-cache that uses the asynchronous version of the dm_io() function. This function does not block so it has a chance to bundle I/O requests and provide better performance, however, at the risk of providing less reliability.

5.1 Evaluation Setup

In order to evaluate our system we require at least three machines; one for each of the roles mentioned in the design section. Unfortunately, due to time constraints we weren't able to acquire three physical nodes to perform our tests, so we decided to run our benchmarks in in virtual machines. There is one virtual machine for each of our roles: a replicator, a replica client, and a storage server. For the first configuration we only used the replicator VM which ran a version of the original dm-cache. The second and third configuration were setup so that both replicator and replica client had access to a storage device in the storage server VM. The replicator VM would then access the replica cache device which was made available by the replica client VM.

All configurations of dm-cache ran using the same set of parameters. The capacity of the source device 5 GB and the cache devices (both local and replica) are of size 1 GB. The data transfer accross all machines is done through the iSCSI protocol. The tests we performed were done using the IOzone benchmarking tool. We used it to write a 128 MB file on the dm-cache client running on the replicator VM.

5.2 Results

We gathered data for cache write miss operations which are one of the cases in which dirty data is written to the cache devices. Figure 2 shows the performance for each of the configurations.

As we can see the results show that in comparison with baseline case, the configuration of dm-cache that uses the asynchronous dm_io operation performs slower, a reduction in 40As for the synchronous dm_io function, the overhead in comparison to the original configuration results in a reduction in performance by up to 65

There are several causes behind the overhead of the replicating configurations of dm-cache. Besides having to write to more than one device, the reduction in performance for asynchrnous operations is most likely caused by the iSCSI communication layer. In other words, in order for the data to be replicated to another peer client, it must travel as a bio to the iSCSI layer, and then be converted to an object which can be transferred across a TCP/IP connection, and then this object must be converted back to a bio which must be handled by the replica client. By finding another more optimal way to replicate data accross clients, this overhead could be reduced. For the synchronous dm_io operations, the previous overhead still applies, but in addition, the slower throughoput is a direct result of the behavior of synchronous operations which will block until each request completes; one at a time.

Lastly, as part of the evaluation, we should state that we are aware that running performance benchmarks in a virtualized environment is not a good representation of the real physical hardware. Nevertheless, we believe that a similar behavior wouldi be observed if our tests were run in physical hardware.

References

[1] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pitchin, A., Sivasubramanian, S., Vosshali, P., and Vogels, W. Dynamo: Amazon's Highly Available Key-value Store. In *18th Symposium on Operating Systems Principles* (2007).

[2] Ford, D., Labelle, F., Popovici, F. I., Stokely, M., Truong, V.-A., Barroso, L., Grimes, C., and Quinlan, S. Availability in globally distributed storage systems. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation* (2010).

[3] Hensbergen, E. V., and Zhao, M. Dynamic Policy Disk Caching for Storage Networking. Tech. Rep. RC24123 (W0611-189), IBM Research, November 2006.

[4] Lei, M., Vrbsky, S. V., and Zijie, Q. Online Grid Replication Optimizers to Improve System Reliability. In *Proceedings of the International Parallel and Distributed Processing Symposium* (2007).

[5] Saito, Y., and Shapiro, M. Optimistic Replication. *ACM Computing Surveys, Vol. 37, No.1* (2005).

[6] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications.

[7] YU, H., AND VAHDAT, A. Minimal Replication Cost for Availability. In *21st Symposium on Principles of Distributed Computing* (2001).

[8] YU, H., AND VAHDAT, A. The Costs and Limits of Availability for Replicated Services. In *18th Symposium on Operating Systems Principles* (2001).

[9] ZHAO, M. Visa research lab: dm-cache. Accessed September 29, 2012.