

Replication-based Highly Available Metadata Management for Cluster File Systems

Zhuan Chen^{*†‡}, Jin Xiong^{*}, and Dan Meng^{*}

^{*} National Research Center for Intelligent Computing Systems
Institute of Computing Technology, Chinese Academy of Sciences

[†] Graduate University of Chinese Academy of Sciences
{chenzhuan, xj, md}@ncic.ac.cn

Abstract—In cluster file systems, the metadata management is critical to the whole system. Past researches mainly focus on journaling which alone is not enough to provide high-available metadata service. Some others try to use replication, but the extra latency accompanied is a main problem. To guarantee both availability and efficiency, we propose a mechanism for building highly available metadata servers based on replication, which integrates Paxos algorithm effectively into metadata service. The Packed Multi-Paxos is proposed to reduce the latency brought by replication, which is self-adaptive and can make the replication to achieve high throughput under heavy client load and low latency under light client load. By designing efficient architecture and coordination mechanism, all replica server nodes simultaneously provide metadata read-access service. This high-available mechanism could decrease the impact of server failures and there is no interruption of service. The performance results show that the latency caused by replication and redundancy is well under control, and the performance of metadata read operation gains improvement.

I. INTRODUCTION

Cluster has become the effective solution to meet the increasing demand of scalable computing and mass-data storage. With ever larger clusters emerging on the market every year, maintaining high levels of robustness and availability has posed a substantial challenge for cluster design. One particularly central concern is the availability of cluster's storage system. Technology and market trends over time may combine to increase the frequency of failure occurrence in storage systems [1]. At the same time, the demand to high-available storage systems and long-time stable I/O service is increasing rapidly in many fields. Failures can be expensive; for instance, in Internet business companies, millions of dollars per hour are lost when storage systems are not available [2]. The availability of storage systems has become a critical problem and gained more and more attention in both academic and industrial areas [3] [4] [5].

Cluster file systems play a central role in cluster's storage management and I/O service. The architecture of today's cluster file systems usually involves the decoupling of metadata processing from the file I/O operations. The metadata servers manage the file system namespace and file attributes, while the data servers process the data read and write operations.

Our previous work [6] built high-available mechanism for data servers. In a cluster file system, the access to data servers are guaranteed by the normal work of metadata service, since the metadata servers maintain all the files' meta-information and define how file system access user data through the storage space. Metadata servers are very important to a file system. The disruption of their service could result in the service downtime of the entire file system and even lead to data loss. A set of highly available metadata servers is critical for building robust cluster file system. This paper aims at this problem.

To maintain the availability of metadata servers, previous researches focus on the technique of journaling [7] [8] [9] [10] [11] [12] [13]. Their idea is to provide atomic metadata operations and thus to maintain the consistency of metadata after failures. With the guarantee of such consistency, metadata servers are able to provide correct service. However, journaling alone has its inherent limitations. It usually doesn't prevent the loss of metadata and state during failures and thus cannot provide seamless recovery and fail-over of metadata service.

Replication is an efficient method for fault-tolerance. With redundant components, when one replica fails, the others are still be able to work. Although the techniques and related issues has been studied before [14] [15] [16], replication is not fully used for metadata servers and most systems' strategies still fall into the category of journaling. Some projects try to explore this topic [17] [18] [19]. One problem encountered is the inevitable latency introduced by replication, since every modification should be copied to all the replicas. Most of these projects do not fully address this problem.

In a file system, metadata-based transactions can account for over 50% of the total number of I/O requests [20] [21]. The efficiency of metadata access and management is critical to the overall performance of the file system. Thus, availability and efficiency are the two key points that should be considered for building stable and robust metadata servers.

This paper presents a practical solution to address these issues. We build a metadata management framework that uses replication for ensuring high availability in cluster file systems. The Paxos algorithm [22] [23] is adopted and the Packed Multi-Paxos method is proposed to reduce the replication latency. This packing mechanism is self-adaptive, which can make the replication achieve high throughput under heavy client load and low latency under light client load. By de-

[‡] Zhuan Chen is currently the graduate student at Department of Computer Science, University of Rochester (zchen@cs.rochester.edu).

signing efficient architecture and coordination mechanism, all replica server nodes simultaneously provide metadata read-access service. The experimental results show that our system can improve the availability of metadata servers through replication while maintaining good performance.

Our work makes the following contributions. (1) We adopt replication to construct the framework for highly available metadata servers, which differs from the traditional technique of journaling used by most of previous work. Our system can reduce the impact of metadata server failure without service interruption or loss. (2) We integrate the Paxos algorithm into the replication scheme for metadata management. With solutions to minimize the cost and latency brought by replication and consensus, this high-available mechanism is much more practicable. (3) By designing efficient architecture and coordination mechanism, we make full use of the resources of all replica server nodes. The performance of metadata read operation increases, as all redundant servers can provide metadata read-access service.

The remainder of this paper is organized as follows. We begin by discussing related work in Section II. Then we present the architecture of this high-available framework in section III and introduce the consensus maintenance for replicas in section IV. Experiment and evaluation are given in section V. Finally we conclude our work in section VI.

II. RELATED WORK

Previous methods mainly focus on journaling to maintain the availability of metadata servers. The introduction of journaling is to solve the problem of metadata inconsistency caused by system crash [24] and to avoid the long-time scavenging of checkers like *fsck* [25]. The technique of journaling is initially used in database systems [26] and local file systems [27] [28] and then extends to distributed file systems [7] [8] [9] [10] [11] [12] [13]. With distributed environment, many systems further put their metadata and log records into shared storage and make them accessible to all metadata server nodes. With such configuration, when one node fails, the remaining nodes are still able to recover the crashed metadata by using the failed one's log records and then take over its service. However, journaling alone has its inherent limitations. To ensure the performance of metadata processing, the log records and metadata are usually written to disk in asynchronous way. This may lead to the loss of memory metadata and state during failures. Besides, the architecture of shared storage may also require special hardware support. Journaling aims at the fast recovery of metadata consistency within each metadata server, which is not the major issue for us. By adopting replication into metadata servers, our system provide seamless recovery and fail-over of metadata service and also minimize the loss of memory metadata and state when failures occur.

Several systems have used the method of versioning. Elephant file system [29] retains all important versions of user files and provides a range of versioning options for recovery from user error. Some systems support snapshots to assist recovery from system failures [30] [31]. CVFS [32] also tries

to decrease the volume of the versioned metadata which could consume as much space as versioned data. In general, the method of versioning also aims at the recovery of metadata consistency from failure within each metadata server. Thus it could share the same drawbacks with journaling.

Checksums are used to detect disk corruption by different systems [33] [34] [35]. Usually, checksums are written to disk with file blocks and are verified when read. If error is detected and mirror or parity check is available, correction and recovery can be carried out. The technique of checksums aims at disk integrity. It is usually implemented in the low-level local file systems [1] and might not be the effective method for the high-level distributed I/O management – such two-level hierarchy is the prevalent architecture for today's many distributed file systems [13] [10] [12] [19] [36].

There is also some research that uses replication for ensuring high availability in metadata servers. Ou *et al.* present the architecture of symmetric active/active metadata service in storage system [17]. This method requires each metadata write request received to be broadcasted and thus sets up a total order of all metadata write requests in the system. The total order could not be established until each metadata server receives the information broadcasted by all other servers. It could introduce large amount of network transmissions and latency. Different from this work, our method adopts Paxos algorithm to do the replication and to maintain the consensus among all replicas. We also propose the Packed Multi-Paxos to further reduce the amount of network transmissions introduced by replication and consensus.

Farsite [18] replicates directory metadata among a set of machines by employing Byzantine-fault-tolerant protocol [37]. To reduce the high cost of Byzantine replication, the updates to metadata are not replicated to all these machines immediately. Instead, all the updates are written into a log and pushed back to this set of machines periodically. In contrast, by controlling the cost, our system could replicate every metadata update to all the replica nodes before replying the result to client.

Google File System (GFS) [19] also adopts replication for its metadata server (called *master* in GFS). The master's operation log and checkpoints are replicated on multiple machines. There are some "shadow" masters (not mirrors) providing read-only access to some files in some situations. No further information about this part is given in GFS's paper and several key points are not revealed, for example, how to do replication among master and its backups, how to maintain consensus among replicas, how to reduce the cost of replication, and so forth. Besides these, our work differs from GFS in that all the metadata servers in our system could provide read-only metadata service, which enables the file system to make full use of the resources of all replica nodes.

The method used by Archipelago [38] is also related with replication. Archipelago divides the nodes in the system into *islands* and replicates metadata skeleton to each island with different degrees. Its idea is to provide failure isolation, that is, although some client requests cannot be served due to the failed islands, it tries to maximize the utilization of the remain-

ing islands. In contrast, we emphasize the high availability of metadata service and achieve this goal by providing full redundancy of metadata information.

III. ARCHITECTURE OF REDUNDANCY FOR METADATA SERVERS

This work is carried out on our DCFS3 (Dawning Cluster File System v3), which is a high-performance distributed file system designed for large-scale clusters. The architecture of DCFS3 consists of three parts: metadata server, object-based storage device, and client. The managements of metadata and data are processed separately, and the client provides the file system interface to users.

We adopt replication to the metadata servers and implement redundancy to improve the availability. This section introduces the design of the redundancy architecture, the method of doing replication for metadata, and how the metadata servers detect and handle the failures based on such redundancy architecture.

A. Asymmetric Architecture for Replicas' Service

Based on the original system of DCFS3, we extend each metadata server to a group of replica nodes. We build the redundant replicas based on the idea of state machine [39], that is, each replica starts from the same state, receives the same inputs, and thus achieves the same new state and outputs. In our system, all the replicas in a group have the same metadata management mechanism. To build them as the same state machine, the same inputs should be replicated to all of them. Among the inputs, what we concern is the metadata write requests, because only write requests could change the state of metadata server.

Different roles are defined for the members in a group. We assign one replica node to be the *primary* and the others to be the *secondaries*. The primary is in charge of the replication of metadata update and order to the secondaries. Specifically, we do this by building an asymmetric architecture.

As shown in Fig. 1, the primary possesses multiple service threads. These threads could handle the incoming metadata requests in concurrent way. Every time each service thread fetches one request from the request queue and processes it separately. The metadata lock is used to handle the conflict among metadata requests and the concurrent metadata modifications. This enables the primary to have the maximum concurrency of metadata operations. As the primary processes the metadata write requests, it obtains the sequence of these requests executed by itself. The order of the metadata write requests then can be defined according to such sequence. The primary sends all the metadata write requests along with their order to the secondaries. While replicating one request to the secondaries and waiting for the reply, the primary can process next requests at the same time. This is also the benefit of multiple service threads.

On the other hand, each secondary has single service thread. With one service thread, the execution of metadata requests is taken under the sequential way. By receiving the information of metadata write requests from the primary, what

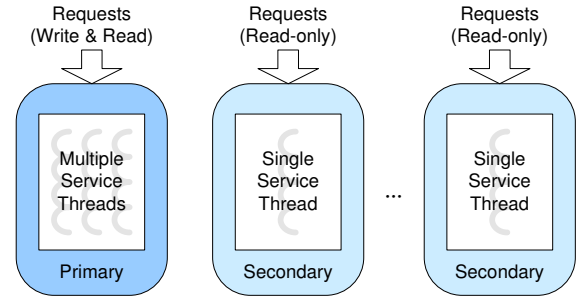


Fig. 1. Asymmetric architecture for replicas' service

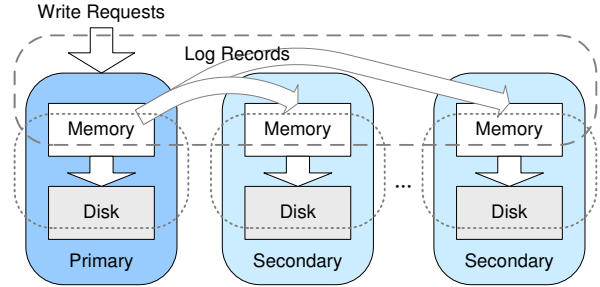


Fig. 2. Replication based on memory metadata and log records

the secondaries need to do is to process these requests under the order provided by the primary. Although the secondaries do not process metadata requests concurrently, they do not have to handle the conflict among multiple metadata requests.

By this way, the clients send metadata write requests to the primary and send metadata read requests to any of the replica nodes, either the primary or the secondary. Within a group, all the metadata write requests and their execution orders are the same for all replica nodes. Each metadata read request is executed separately and locally on the single node. Multiple metadata servers can process concurrent read-only requests simultaneously (this needs some extra coordination which will be discussed later).

B. Replication Based on Memory Metadata and Log Records

Metadata operation is a kind of transaction which consists of a series of basic sub-operations. There might be relevance and interaction among these basic sub-operations. The process of metadata operations should fulfill the properties of transactions [40]. To guarantee the correctness of metadata modifications in both primary and secondaries, we replicate metadata by replicating metadata operations.

The process of replication is as shown in Fig. 2. The primary encapsulates each metadata write operation into a log record and send it to the secondaries. A log record contains the type, arguments, and results of the corresponding metadata write operation. A log number is attached to define the order of each operation. Based on the information of the log records, the secondaries directly apply the results of the corresponding operations to their own metadata. By this way, the results of the metadata write operations are the same in both primary and secondaries. Moreover, by simply applying the results from log

records to metadata, the secondaries do not need to repeat the whole process of the metadata write operations, which would make the execution faster.

Journaling is also adopted in our system to maintain the metadata consistency within each single metadata server. This could make the rebuilding of metadata consistency for failed node more efficient. Journaling requires recording log records for metadata write operations. Since we have set up such log records in the primary, they can be directly used for journaling mechanism. The replication of log records among replica nodes also makes it unnecessary for the secondaries to do the encapsulation of log records again.

Once the log records have been replicated to all the secondaries, the primary is able to return the results of the corresponding operations. Each secondary then separately apply these operations and commit to disk without considering the interaction with other replica nodes. The replication would be quick, since we only need to guarantee the consistency of replica nodes at the level of memory and only for the completion of replication. We can do this because by using log records, all replica nodes will have the same metadata update results; thus, to build them as the same state machines, we only need to ensure that they have the same inputs.

C. Failure Detection and Handling

The number of metadata servers in a file system would not be very large, so is the size of each group of redundant metadata servers. We use *heartbeat* [41] to do failure detection. In a group, the primary periodically sends heartbeat message to the secondaries. The secondary is regarded as failed if it cannot reply within a certain time. On the other hand, if the secondary haven't received the heartbeat message from the primary for a certain time, we would assume the primary is failed. By this way, the failure of the primary or the secondary can always be detected.

If the primary fails, one secondary will finally detect it. It then starts a process of primary election which will generate a new primary. Any clients who have sent requests to the original primary without receiving replies will try to contact other server to get the new view information. The view information contains the current formation about this group. The clients then re-send the requests to this group's new primary.

If the primary detects the failure of a secondary, it just removes this secondary from group's view information. Any clients who have sent requests to this secondary without receiving replies will try to contact another server to get the new view information. They then re-send the requests to any replica node of this group (because the metadata requests sent to the secondaries are always read-only and they can be sent to any replica node, either the primary or the secondary).

IV. CONSENSUS OF REPLICAS FOR METADATA SERVERS

Based on the above redundancy architecture, our system is able to provide high-available metadata service. Besides, we still need to consider the consensus problems under the occurrence of failures. We integrate the Paxos algorithm to

the process of metadata replication to maintain the consensus among replica nodes.

In this section, we introduce the problems for Paxos algorithm implementation and the methods we adopt to address them. We also describe how we use these mechanisms to handle the consensus processing in our system.

A. Problems for Paxos Algorithm Implementation

Paxos algorithm [22] [23] is an efficient protocol to achieve consensus for replicated state machines. It can be executed by a set of processes to agree on a single value despite the existence of failures. To guarantee progress, a distinguished process is always selected as the only one to try issuing proposals. We call this distinguished process as *primary* and other processes as *secondaries*. This organization coincides with that in our redundancy architecture.

Paxos algorithm consists of two phases. (1) In phase 1, a round number would be proposed by the primary, and the secondaries are queried to learn their status for past rounds. (2) In phase 2, the primary chooses a value for this round and tries to get a majority to accept it. These two phases are carried out through the transmission of two kinds of message, the *prepare* message and the *accept* message, respectively. If the second phase is successful, the value would be distributed as the outcome to everyone by broadcasting the *success* message. The process of these two phases is called Basic Paxos. Within a Basic Paxos, we can reach consensus on one value.

Instead of reach consensus on one value, practical systems usually have to obtain consensus on a sequence of values and their order. The simple way to do this is to repeatedly execute the Basic Paxos, which allows only one Basic Paxos instance to be processed one time before another. Some systems use this strategy [42] [43]. For each basic Paxos, there are three kinds of message-passing delay as mentioned above. An optimization along with this algorithm is called Multi-Paxos [22]. If the primary identity does not change between instances, Multi-Paxos allows a newly chosen primary to execute phase 1 once for infinite many instances and then to repeatedly execute phase 2 for the following instances. Since the failure of primary should be rare events, the effective cost is only the phase 2 which has the minimum possible cost of any algorithm for reaching agreement in the presence of faults [44]. However, Multi-Paxos still involves two kinds of message delay, the *accept* message and the *success* message. The latency still poses large amount of cost burden to the implementation of this consensus algorithm.

B. Packed Multi-Paxos

To make the implementation of Paxos algorithm effective and practicable, the cost brought by extra network transmission should be controlled. The latency of Multi-Paxos mainly comes from the message transmissions in phase 2, that is, the *accept* message and the *success* message. Our strategy is to reduce the latency caused by these two kinds of message.

The latency brought by replication is inevitable. For the analogous problem in data servers, Google file system replicates data among multiple chunkservers (its data servers) and

pushes data in a pipelined fashion [19]. While this strategy can fully utilize each data server’s network bandwidth and minimize the extra latency, it is not suitable for metadata servers. Metadata is in general small and the replication of metadata among different servers usually could not take full use of the network bandwidth. The demand of entire network transmission for each metadata replication would produce large amount of extra latency.

Our idea is to pack several metadata log records together and to transmit them within one message-passing process. The problem is how to decide the number of log records to be packed and how often to do the packing. Intuitively, the more log records we put into one message, the less network transmissions are needed for replication. However, to pack a certain amount of log records into one message, this message cannot be sent until all of these log records have been generated. This might increase the response latency for the corresponding metadata write requests, especially when the client load is light. Some certain parameters are suggested [45], but fixed configuration is not flexible. To handle this tradeoff, we build the packing mechanism to be self-adaptive, which can make the replication achieve high throughput under heavy client load and low latency under light client load.

We propose Packed Multi-Paxos to decrease the transmission number of *accept* messages. As shown in Fig. 3, we introduce packing thread into the primary. The primary now possesses several service threads and several packing threads; metadata operations and replications are carried out under the interaction of these threads. Multiple packing threads are coordinated by a lock, which guarantees that at any time only one packing thread can get the lock and be active. Service threads do not directly replicate the log records to the secondaries. Instead, after the encapsulation of log records, they notify the active packing thread. Once notified, the active packing thread will pack all the log records existed in current system and replicate them to the secondaries through one network transmission. Replication through network is asynchronous. The active packing thread releases its lock after the message has been sent out; while it is waiting for the reply, another packing thread can get the lock and become active. The asynchronous way could improve the performance and that is why we use multiple packing threads.

The self-adaptive property is achieved through the help of thread scheduling mechanism in operating system. Service threads and packing threads have the same priority and are scheduled equally. If the current client load is heavy, before the active packing thread is scheduled, several or nearly all service threads might have already been scheduled. Thus there could be multiple log records existing in system before the packing thread tries to pack them. The heavier the load, the more log records are likely to be packed into one network transmission. On the other hand, if the current client load is light, even when there is only one service thread generating log record, the active packing thread will be scheduled very soon and then send this log record immediately. This is because there is no metadata processing task for other service threads, so

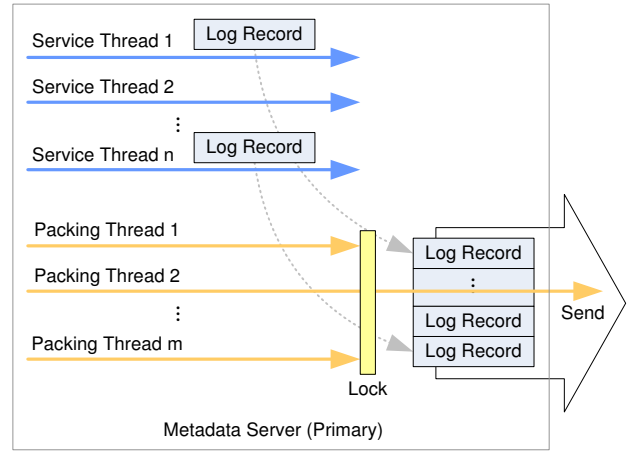


Fig. 3. Packed Multi-Paxos

they are all asleep (at the same time the other packing threads are also asleep); the scheduling mechanism will just bypass them. By utilizing the operating system’s thread scheduling, the number of log records within each message can be self-adaptive according to the current system load.

The packing mechanism may rely much on the operating system scheduler. The scheduler often schedules threads in a round-robin manner, so how many log records can be packed together is also determined by how long a time slice is and how quickly a log record is generated. Metadata is small and its update is always quick, so is the generation of log record. Besides, for the operating system, the time slice for round-robin scheduling is usually not too short, otherwise the cost for scheduling would be high. As a result, our method based on operation system scheduling could work. We will give the specific evaluation later.

Above is about the reduction of the *accept* message’s transmission number. For *success* message, which is used to notify the secondaries that the replication is successful, we simply embed it into the next *accept* message to further reduce the number of message passing. Note that the *success* message should also be put into the primary’s request queue, because after that there may be no *accept* message any more. We also guarantee that finally the same *success* message will be sent only once.

C. Paxos Coordination Queue

Although we pack several log records into one message, each log record is in a separate process of Basic Paxos. We need to maintain their own state. Besides, for Packed Multi-Paxos, we also need to maintain the order of each log record in the secondaries. We build Paxos Coordination Queue (PCQ), which is a data structure to manage and coordinate the log records in the secondaries. We also keep PCQ in the primary, in case its role change.

When the secondary receive a log record, it does not apply the corresponding metadata operation immediately. Instead, it puts this log record into PCQ. As shown in Fig. 4, PCQ is a data structure which consists of log records. Each log

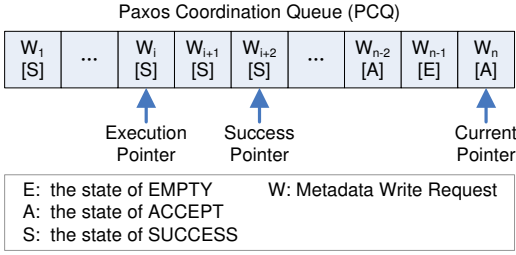


Fig. 4. The structure of Paxos Coordination Queue

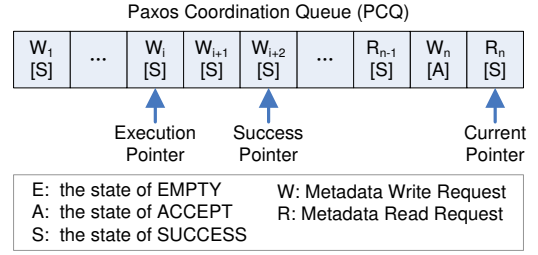


Fig. 5. The records of metadata write and read requests in PCQ (in secondary)

record in PCQ has three types of state: *EMPTY*, *ACCEPT*, and *SUCCESS*. (1) A log record has the state of *ACCEPT* at the time when it is put into PCQ. All records in PCQ are arrayed by their log numbers. (2) The log numbers should be continuous. If not, the interval among them should be filled by empty ones which have the state of *EMPTY*. (3) A record will be changed to the state of *SUCCESS* when the secondary receives the corresponding *success* message, which denotes that the log record has been accepted by the majority. The secondaries cannot execute a log record until this log record has the state of *SUCCESS*.

The secondary executes metadata write operations by the sequence arrayed in PCQ. PCQ also contains three kinds of pointers to manage the execution order. (1) The *Current Pointer* points to the latest added log record. (2) The log record pointed by the *Success Pointer* and its former log records all have the state of *SUCCESS*. (3) The *Execution Pointer* points to the record that has already been executed by the secondary, and meanwhile, its former log records have also been executed.

PCQ manages and coordinates the execution of metadata operations. Besides this, PCQ is used to coordinate the relevance between the metadata write requests and read requests for the secondaries. PCQ is also involved in the system recovery after failure. We describe these next.

D. Handling Metadata Read Requests

In our system, all the replica metadata servers can process read-only requests simultaneously. The handling of metadata read requests is simple for the primary. It just puts the metadata read requests into the request queue, and executes all metadata requests from the request queue without distinguishing between write request and read request.

However, the handling of metadata read requests for the secondaries is nontrivial. The secondaries receive the metadata write requests from the primary and receive the metadata read requests from the clients. They come from different sources and possess different execution patterns. There might also exist some relevance between these two kinds of metadata requests, for example, one particular write request should be done before the other particular read request. To handle this, we also use PCQ to guarantee the correct execution of metadata requests in the secondaries.

As shown in Fig. 5, every time when the secondary receives a metadata read request, it puts it to the tail of PCQ as a record. The record then immediately has the state of *SUCCESS*, since

the corresponding metadata read request is able to be executed and will be handled locally. The record of read request in PCQ has the same log number as that of the previous one. By doing this, it would not disturb the execution order of write requests. The primary would not return the results of a metadata write operation until the corresponding log record has been replicated to every secondary's PCQ. By executing requests based on the sequence arrayed in PCQ, the secondary can guarantee the interacted relationship between metadata write requests and read requests.

E. Recovery

As mentioned before, for the failure of primary, a process of primary election needs to be carried out. In our method, the primary finally elected should be the one possesses the highest log number, that is, the log number of the record pointed by its *Current Pointer* should be the largest.

To do recovery, we introduce the *Recovery Pointer* to the PCQ. In the end of primary election, each secondary reports its log number of the record pointed by its *Success Pointer*. The primary picks the smallest one of these reported log numbers (including that of the primary) and makes its *Recovery Pointer* point to the record with that log number. The primary then will fix the inconsistency of the records from the one pointed by its *Recovery Pointer* to the one pointed by its *Current Pointer*, which will be the recovery range.

During the process of recovery, the primary fetches every record from the *recovery range* and processes them according to different situations. (1) If the record has the state of *SUCCESS*, which means this record has already been accepted by the majority before. The primary simply sends this record to the secondaries. (2) If the record has the state of *ACCEPT*, which means this record has been accepted by the primary itself before, but we have no idea whether it has been accepted by the majority. The primary tries to ask the majority to accept this record by sending the *accept* message, which is the same as the phase 2 of Paxos algorithm described before. (3) If the record has the state of *EMPTY*, the primary should try to gather the information about this record by processing the phase 1 of Paxos algorithm. At this stage, if no one in the group knows about this record, the primary would treat this record as “no-op” [23] which will do nothing to the state of the metadata server. The primary then sends this record of “no-op” to the secondaries.

After the recovery process, all records that may lose their consensus during failure would have been recovered. The consistency among the replica servers is achieved again. The primary then continues to process the metadata requests from the clients, and the metadata service can work normally.

V. EVALUATION

We have developed a prototype implementation of the proposed high-available mechanism. Using our prototype, this section evaluates the performance and the function of our system. For evaluation, the configuration of DCFS3 consists of three metadata servers (MDS), one or two object-based storage devices (OSD), and several clients. We use the nodes created by VMware to do evaluation. Each node contains one Intel Xeon Processor (2.00GHz), 1-Gbyte memory, and 36GB disk space. All of them are interconnected through Gigabit Ethernet (the actual network transmission speed is 764.91 Mbits/sec tested by Netperf [46]).

Our evaluation employs one benchmark and one application:

- *mdtest* [47] [48] is a metadata benchmark for file system. It performs `open/stat/close` operations on files and directories and then reports the performance. *Mdtest* also supports MPI [49]. It could be used to test the performance of metadata servers under the load of intensive metadata operations.
- *mpiBLAST* [50] [51] is a scientific computing application. It is a parallel implementation of NCBI BLAST [52] and can improve the performance of NCBI BLAST by utilizing distributed computational resources. Cluster is a typical platform for *mpiBLAST* and *mpiBLAST* is also a typical application for testing in cluster.

A. Performance

We present the performance of metadata operations by running *mdtest*. The configuration of DCFS3 consists of 1 MDS, 1 OSD, and several clients. The number of clients varies from 1 to 12. The number of service threads in primary is 10. We use 100 threads to run *mdtest* through MPI. Each thread handles `open/stat/close` operations to 1,000 files under its own separate directory. Thus total 100,000 files are involved in this evaluation.

We compare three kinds of mechanism under the same configuration:

- 1) the original metadata server of DCFS3 without fault-tolerance method
- 2) our proposed high-available mechanism with the Packed Multi-Paxos and the PCQ
- 3) the high-available mechanism with the standard Multi-Paxos algorithm

Fig. 6 shows the operation of *CREATE* performed by *mdtest*. This represents the performance of metadata write operations. Based on the original DCFS3 without any fault-tolerance, we can evaluate the performance degradation of the fault-tolerant methods. Compare to the original metadata server of DCFS3, our high-available mechanism only has a performance degradation of 1/4, which is much better than

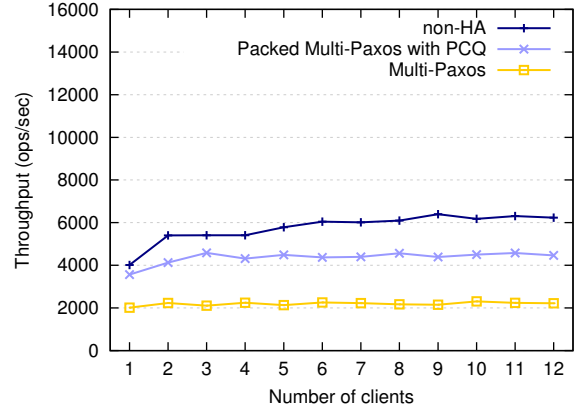


Fig. 6. Performance of metadata write operation (*CREATE*)

TABLE I
NUMBER OF NETWORK TRANSMISSIONS WITHIN THE GROUP OF REDUNDANT MDSS

Number of log records contained in network transmission	Number of network transmissions within the group of redundant MDSS		
	1 MDS, non-HA	3 MDS, HA Multi-Paxos	3 MDS, HA Packed Multi-Paxos with PCQ
0	0	300300	304
1	0	300300	764
2	0	0	3134
3	0	0	11136
4	0	0	20403
5	0	0	19476
6	0	0	10148
7	0	0	2411
8	0	0	336
9	0	0	45
10	0	0	1
Total	0	600600	68158

that with the the standard Multi-Paxos (about 2/3 performance degradation). The result shows that the adoption of Packed Multi-Paxos is able to decrease the impact brought by replication.

In our configuration, the primary possesses 10 service threads, thus the number of log records contained in one message could be in the range from 0 to 10. Among them, the number of 0 denotes the network transmission containing no log record but only the *success* message. Here we analyze the number of network transmissions within a group of redundant metadata servers. As shown in TABLE I, the original metadata server of DCFS3 has no such network transmission since it has no replication. The high-available mechanism with standard Multi-Paxos includes two types of message passing, the *accept* message and the *success* message. The former one contains one log record while the later one contains none. For the packing mechanism used in our system, the number of log records packed in one message varies from 0 to 10. Among them, the ones containing 4 and 5 log records have the maximum volume. Compared with the standard Multi-Paxos,



Fig. 7. Performance of metadata read operation (STAT)

our packing mechanism can reduce the number of network transmissions by about 8/9.

Fig. 7 shows the operation of *STAT* performed by *mdtest*. This represents the performance of metadata read operations. In our system, all redundant metadata servers could provide read-only service and it has much better performance than that of the original metadata server of DCFS3. This is because multiple servers are able to process concurrent metadata read requests simultaneously.

B. Service Continuity

We also use *mdtest* to test the function of fault-tolerance in our system. During the running of *mdtest*, we turn off one node to see whether the process of *mdtest* could finish. The configuration of DCFS3 consists of 1 MDS, 1 OSD, and 3 clients. We use 30 threads to run *mdtest* through MPI. Each thread will do operations to 1,000 files under its own separate directory. Thus total 30,000 files are involved in this evaluation.

We present the performance of high-available mechanism during the period when failure occurs. The running of *mdtest* is performed for five times. At the 3rd time, we turn off one metadata server node (primary or secondary) when *mdtest* is doing the operation of *CREATE*. Since there are still majority existing, the metadata service should be able to continue.

Fig. 8 shows the situation when the primary is turned off at the 3rd *mdtest* running. In Fig. 8 (a), the result shows that the metadata service is able to continue. For the 3rd *mdtest* running, the performance of *CREATE* operation drops, this is due to the process of fail-over after the failure. The performance of *CREATE* operation at the last two times is better than that of the first two times. This is because after the primary is turned off, the number of members in the group is reduced by one; the new primary only has to broadcast the metadata write requests to one secondary. Accordingly, the performance of *STAT* operation drops after the 3rd running, since the failure decreases the number of metadata servers which are able to provide metadata read service.

Fig. 8 (b) shows the time spent by each *mdtest* running. The time used at 3rd running is longer than that of the others as it

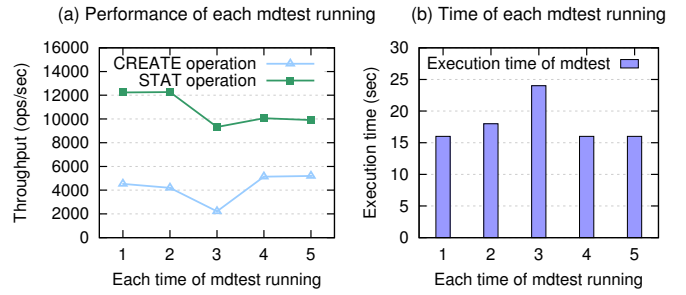


Fig. 8. Comparison for performance and time spending (primary failure)

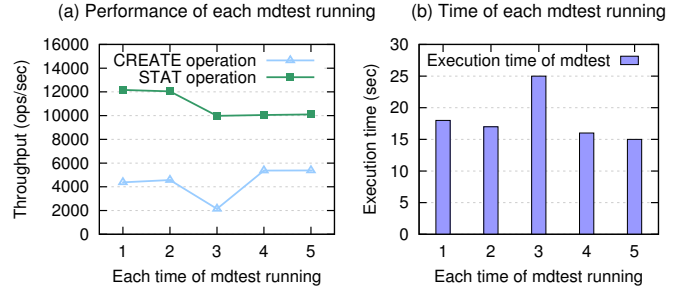


Fig. 9. Comparison for performance and time spending (secondary failure)

involves the process of fail-over. The time spent for fail-over consists of two aspects, for evaluation, (1) the timeouts of the clients to wait for reply is 5 seconds, and the clients would wait for another 2 seconds before requesting for new view information from metadata server; (2) the timeouts of waiting for heartbeat in metadata server is 2 seconds, and the process of primary election and recovery is less than 10 milliseconds by our test. As a result, the time spent for fail-over is mostly due to the timeouts of waiting. The process of recovery is fast.

Fig. 9 shows the situation when one secondary is turned off at the 3rd *mdtest* running. The results are almost the same as that shown in Fig. 8. Different from primary failure, there is no need for primary election and recovery when secondary fails. The primary just need to remove the failed secondary from its view and then can continue the service. Nearly all the time spent for fail-over is the timeouts, which demonstrates again that the most latency of fail-over is due to the timeouts of waiting. In summary, when failure occurs, our system has a fast recovery and is able to provide continuous I/O service.

C. mpiBLAST Application

Cluster is a typical platform for mpiBLAST. Running mpiBLAST is an efficient way to evaluate the performance of our system and its support to application. The configuration of DCFS3 consists of 1 MDS, 2 OSD, and the numbers of clients are 3, 7, and 11 for each three tests. The size of database used is 977 MB. We divide it into several segmentations according to the demonstration of FAQ [51], as shown in TABLE II.

The first step of mpiBLAST is database segmentation. It then processes the gene query and matching, and next predicts the characteristic of the unknown protein. In the end,

TABLE II
CONFIGURATION OF DATABASE SEGMENTATION AND MPI THREAD
NUMBER IN MPIBLAST

Number of Clients	Number of Database Segmentations	Number of MPI Threads
3	2	4
7	6	8
11	10	12

mpiBLAST would display the time spent for the process of query and matching. We do the evaluation based on this time.

Fig. 10 shows the performance of mpiBLAST running under two situations, the original DCFS3 without fault-tolerance and the one with our high-available mechanism. The result shows that there is no distinct difference for the running time (less than one second) between these two situations. In mpiBLAST, the impact of extra latency brought by our high-available mechanism is negligible.

Fig. 11 shows the impact of failure to mpiBLAST running. We turn off one server node (primary or secondary) when mpiBLAST is doing gene query and matching. With node failure, the process of mpiBLAST can still finish, which means the metadata service is able to continue despite the failure. The running time is just a bit longer than the situation when there is no failure. Either under the situation of primary failure or secondary failure, the time spent for fail-over has no distinct difference. Such conclusion is the same as that of mdtest.

VI. CONCLUSION

The availability of storage systems has become a central problem in cluster design. Within the file system, a set of highly available metadata servers is a critical component for building robust cluster file system. Previous work mainly focuses on journaling which alone is not enough to provide high-available metadata service. Replication is adopted by some systems, but the controlling of replication latency is a main problem. Considering two key points, availability and efficiency, we propose a high-available mechanism for metadata service based on replication. To implement Paxos algorithm effectively into the high-available mechanism, the Packed Multi-Paxos is proposed to reduce the replication latency. The proposed solution can effectively improve throughput and reduce latency under different client load. The Paxos Coordination Queue (PCQ) is also designed to manage and coordinate the log records and to assist the process of metadata management and replication. The high-available mechanism can decrease the impact of server failures and there is no disruption of service. While the latency of replication is well controlled for the metadata write operations, our system also enables the performance of metadata read operations to gain improvement.

ACKNOWLEDGEMENT

This work was supported in part by the Natural Science Foundation of China under grant numbered 60970025, and the National High-Technology Research and Development

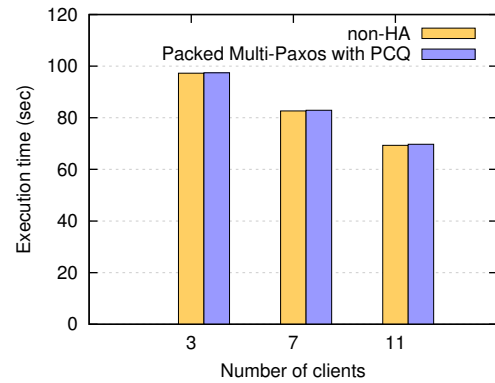


Fig. 10. Comparison of non-HA and HA in mpiBLAST

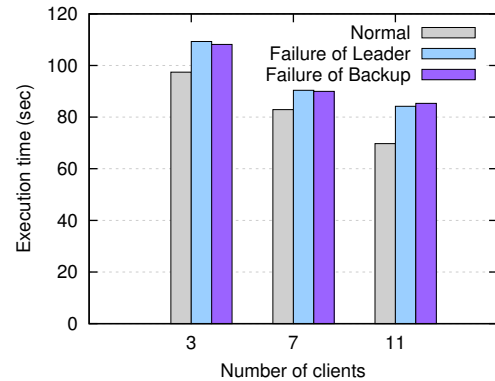


Fig. 11. The impact of failure to mpiBLAST running

Program of China under grant numbered 2006AA01A102, 2009AA01Z139 and 2009AA01A129. We would like to thank the anonymous reviewers for the detailed comments and suggestions that helped improve this paper.

REFERENCES

- [1] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Iron file systems," in *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*. New York, NY, USA: ACM, 2005, pp. 206–220.
- [2] D. Patterson, "Availability and maintainability \hat{u} performance: New focus for a new century," *Key note speech at FAST '02*, 2002.
- [3] B. Schroeder and G. A. Gibson, "A large-scale study of failures in high-performance computing systems," in *DSN '06: Proceedings of the International Conference on Dependable Systems and Networks*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 249–258.
- [4] —, "Disk failures in the real world: what does an mttf of 1,000,000 hours mean to you?" in *FAST '07: Proceedings of the 5th USENIX conference on File and Storage Technologies*. Berkeley, CA, USA: USENIX Association, 2007, p. 1.
- [5] E. Pinheiro, W.-D. Weber, and L. A. Barroso, "Failure trends in a large disk drive population," in *FAST '07: Proceedings of the 5th USENIX conference on File and Storage Technologies*. Berkeley, CA, USA: USENIX Association, 2007, pp. 2–2.
- [6] L. Cao, Y. Wang, and J. Xiong, "Building highly available cluster file system based on replication," in *PDCAT '09: Proceedings of the 2009 International Conference on Parallel and Distributed Computing, Applications and Technologies*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 94–101.
- [7] A. Bhide, E. N. Elnozahy, and S. P. Morgan, "A highly available network file server," in *In Proceedings of the Winter USENIX Conference*, 1991, p. pages.

- [8] C. A. Thekkath, T. Mann, and E. K. Lee, "Frangipani: a scalable distributed file system," *SIGOPS Oper. Syst. Rev.*, vol. 31, no. 5, pp. 224–237, 1997.
- [9] K. W. Preslan, A. Barry, J. Brassow, M. Declerck, A. J. Lewis, A. Manthei, B. Marzinski, E. Nygaard, S. Van Oort, D. Teigland, M. Tilstra, S. Whitehouse, and M. O'Keefe, "Scalability and failure recovery in a linux cluster file system," in *ALS'00: Proceedings of the 4th annual Linux Showcase & Conference*. Berkeley, CA, USA: USENIX Association, 2000, pp. 10–10.
- [10] I. F. Haddad, "Pvfs: A parallel virtual file system for linux clusters," *Linux J.*, p. 5, 2000.
- [11] J. Menon, D. A. Pease, R. Rees, L. Duyanovich, and B. Hillsberg, "Ibm storage tank— a heterogeneous scalable san file system," *IBM Syst. J.*, vol. 42, no. 2, pp. 250–267, 2003.
- [12] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: a scalable, high-performance distributed file system," in *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*. Berkeley, CA, USA: USENIX Association, 2006, pp. 307–320.
- [13] R. Z. PJ Braam, "Lustre: A scalable, high performance file system," *Cluster File Systems, Inc.*
- [14] Engelmann, Scott, C. Engelmann, and S. L. Scott, "Concepts for high availability in scientific high-end computing," in *In Proceedings of High Availability and Performance Workshop (HAPCW) 2005*, 2005, p. 2005.
- [15] G. V. Chockler, I. Keidar, and R. Vitenberg, "Group communication specifications: a comprehensive study," *ACM Comput. Surv.*, vol. 33, no. 4, pp. 427–469, 2001.
- [16] X. Défago, A. Schiper, and P. Urbán, "Total order broadcast and multicast algorithms: Taxonomy and survey," *ACM Comput. Surv.*, vol. 36, no. 4, pp. 372–421, 2004.
- [17] L. Ou, C. Engelmann, X. He, X. Chen, and S. Scott, "Symmetric active/active metadata service for highly available cluster storage systems," in *In Proceedings of the 19th IASTED International Conference on Parallel and Distributed Computing and Systems*. ACTA Press, 2007.
- [18] 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," in *OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation*. New York, NY, USA: ACM, 2002, pp. 1–14.
- [19] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*. New York, NY, USA: ACM, 2003, pp. 29–43.
- [20] D. Roselli, J. R. Lorch, and T. E. Anderson, "A comparison of file system workloads," in *ATEC '00: Proceedings of the annual conference on USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2000, pp. 4–4.
- [21] A. Traeger, E. Zadok, N. Joukov, and C. P. Wright, "A nine year study of file system and storage benchmarking," *Trans. Storage*, vol. 4, no. 2, pp. 1–56, 2008.
- [22] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, 1998.
- [23] —, "Paxos made simple," *ACM SIGACT News*, vol. 32, pp. 18–25, 2001.
- [24] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel (3rd ed.)*. O'Reilly, 2005.
- [25] T. J. Kowalski, "Fsck—the unix file system check program," pp. 581–592, 1990.
- [26] J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu, and I. Traiger, "The recovery manager of the system r database manager," *ACM Comput. Surv.*, vol. 13, no. 2, pp. 223–242, 1981.
- [27] S. Chutani, O. T. Anderson, M. L. Kazar, B. W. Leverett, W. A. Mason, R. N. Sidebotham, and T. Corporation, "The episode file system," in *USENIX Annual Technical Conference*, 1992, p. 43C60.
- [28] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck, "Scalability in the xfs file system," in *ATEC '96: Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 1996, pp. 1–1.
- [29] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Ofir, "Deciding when to forget in the elephant file system," in *SOSP '99: Proceedings of the seventeenth ACM symposium on Operating systems principles*. New York, NY, USA: ACM, 1999, pp. 110–123.
- [30] S. Quinlan, "A cached worm file system," *Softw. Pract. Exper.*, vol. 21, no. 12, pp. 1289–1299, 1991.
- [31] R. J. Green, A. C. Baird, and J. C. Davies, "Designing a fast, on-line backup system for a log-structured file system," *Digital Tech. J.*, vol. 8, no. 2, pp. 32–45, 1996.
- [32] C. A. N. Soules, G. R. Goodson, J. D. Strunk, and G. R. Ganger, "Metadata efficiency in versioning file systems," in *FAST '03: Proceedings of the 2nd USENIX Conference on File and Storage Technologies*. Berkeley, CA, USA: USENIX Association, 2003, pp. 43–58.
- [33] W. Bartlett and L. Spainhower, "Commercial fault tolerance: A tale of two systems," *IEEE Trans. Dependable Secur. Comput.*, vol. 1, no. 1, pp. 87–96, 2004.
- [34] J. Bonwick and B. Moore. (2006) Zfs: The last word in file systems. [http://opensolaris.org/os/community/zfs/docs/zfs last.pdf](http://opensolaris.org/os/community/zfs/docs/zfs%20last.pdf).
- [35] C. A. Stein, J. H. Howard, and M. I. Seltzer, "Unifying file system protection," in *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2001, pp. 79–90.
- [36] B. S. White, M. Walker, M. Humphrey, and A. S. Grimshaw, "Legionfs: a secure and scalable file system supporting cross-domain high-performance applications," in *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*. New York, NY, USA: ACM, 2001, pp. 59–59.
- [37] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *OSDI '99: Proceedings of the third symposium on Operating systems design and implementation*. Berkeley, CA, USA: USENIX Association, 1999, pp. 173–186.
- [38] M. Ji, E. W. Felten, R. Wang, and J. P. Singh, "Archipelago: an island-based file system for highly available and scalable internet services," in *WSS'00: Proceedings of the 4th conference on USENIX Windows Systems Symposium*. Berkeley, CA, USA: USENIX Association, 2000, pp. 1–1.
- [39] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: a tutorial," *ACM Comput. Surv.*, vol. 22, no. 4, pp. 299–319, 1990.
- [40] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992.
- [41] M. K. Aguilera, W. Chen, and S. Toueg, "Heartbeat: A timeout-free failure detector for quiescent reliable communication," in *WDAG '97: Proceedings of the 11th International Workshop on Distributed Algorithms*. London, UK: Springer-Verlag, 1997, pp. 126–140.
- [42] S. A. Weil, A. W. Leung, S. A. Brandt, and C. Maltzahn, "Rados: a scalable, reliable storage service for petabyte-scale storage clusters," in *PDSW '07: Proceedings of the 2nd international workshop on Petascale data storage*. New York, NY, USA: ACM, 2007, pp. 35–44.
- [43] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou, "Boxwood: abstractions as the foundation for storage infrastructure," in *OSDI'04: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*. Berkeley, CA, USA: USENIX Association, 2004, pp. 8–8.
- [44] I. Keidar and S. Rajsbaum, "On the cost of fault-tolerant consensus when there are no faults: preliminary version," *SIGACT News*, vol. 32, no. 2, pp. 45–63, 2001.
- [45] Y. Mao, F. P. Junqueira, and K. Marzullo, "Mencius: Building efficient replicated state machines for wans," in *Proceedings of the 8th symposium on Operating systems design and implementation*. USENIX Association, 2008.
- [46] <http://www.netperf.org/netperf/>.
- [47] T. T. M. W. E. Loewe, R. M. Hedges and C. Morrone, "Lnl's parallel i/o testing tools and techniques for asc parallel file systems," in *2004 IEEE Cluster Computing Conference*, 2004.
- [48] <http://sourceforge.net/projects/mdtest/>.
- [49] E. L. W. Gropp and A. Skjellum, *Portable Parallel Programming with the Message-Passing Interface*. MIT Press, Cambridge, Mass, 1999.
- [50] A. Darling, "mpiblast evolves: success, collaborations, and challenges," in *Bioinformatics Open-Source Conference 2005 (BOSC'2005)*, 2005.
- [51] <http://www.mpiblast.org/>.
- [52] <http://blast.ncbi.nlm.nih.gov/blast.cgi>.