One Phase Commit: A Low Overhead Atomic Commitment Protocol for Scalable Metadata Services

Giuseppe Congiu
Emerging Technology Group
Xyratex Technology Ltd
Havant, United Kingdom
Giuseppe_Congiu@xyratex.com

Matthias Grawinkel
University of Paderborn
Paderborn, Germany
grawinkel@upb.de

Sai Narasimhamurthy
Emerging Technology Group,
Xyratex Technology Ltd
Havant, United Kingdom
Sai_Narasimhamurthy@xyratex.com

André Brinkmann
Johannes Gutenberg-University
of Mainz
Mainz, Germany
brinkman@uni-mainz.de

***Abstract*—As the number of client machines in high end computing clusters increases, the file system cannot keep up with the resulting volume of requests, using a centralized metadata server. This problem will be even more prominent with the advent of the exascale computing age. In this context, the centralized metadata server represents a bottleneck for the scaling of the file system performance as well as a single point of failure. To overcome this problem, file systems are evolving from centralized metadata services to distributed metadata services. The metadata distribution raises a number of additional problems that must be taken into account. In this paper we will focus on the problem of managing distributed namespace operations such as `CREATE`, `DELETE` and `RENAME`. Distributed namespace operations are a side effect of metadata distribution across the cluster of metadata servers.**

Available protocols for handling distributed namespace operations such as the two phase commitment protocol are expensive since they require the exchange of a large number of messages between metadata servers as well as synchronous writes to stable storage to log vital information. Moreover, such protocols adopt locking schemes to protect the resource during the operation, which force multiple operations on the same directory to be serialized. This severely impacts the performance of high performance computing applications in typical scenarios such as high rate of file create operations.

We propose a one phase commit protocol that is tailored to the use for typical inter-metadata messages. We rely on a fast, highly available shared storage for metadata in order to minimize writes, messages, coordination overhead and recovery time in case of failing metadata servers. We present a formal description of the new protocol, a theoretical analysis of its capabilities, a proof of correctness and the evaluation of the protocol in a simulated environment that renders the protocol to be fast and reliable. In simulations the protocol achieved more than 50% better performance compared with the two phase commitment protocol.

***Index Terms*—computing clusters; exascale computing; distributed metadata; distributed namespace;**

I. Introduction

Upcoming exa-scale computing systems will consist of hundreds of thousands of client machines, which access a shared, distributed file system. Achieving high performance in terms of aggregate application bandwidth and I/O response times is not feasible just by incrementally adding new storage devices as part of existing architectures.

During the last few years, file system designers have tried to achieve better performance and scalability by decoupling the data path from the metadata path and assigning them to dedicated storage and metadata servers (MDS), and by adding more storage devices to scale in capacity and I/O throughput.

This has been possible due to the unstructured nature of data content that enables it to be easily chopped up into stripes and distributed over the available storage devices. Lustre [1] is an example of a file system for large computing clusters that adopts this approach. Nevertheless, as the number of clients accessing the file system scales, managing all the metadata operations using only one centralized MDS, as is common at this time, is very inefficient. The single MDS suddenly becomes a bottleneck for the whole system performance, with the number of requests that can be served per unit time being limited by its capability.

To overcome this problem, a commonly applied solution is to use a cluster of MDSs rather than a single one [2]–[4]. This allows distributing metadata between available MDSs, potentially enabling better scalability (by adding more MDSs when it is required) and availability (if an MDS crashes, the corresponding metadata responsibility can be redistributed amongst the remaining servers). However, unlike data, the scaling of the metadata service is not trivial because the metadata information has a highly structured nature that makes its distribution policy over the servers in the cluster a crucial design issue, which has to be carefully evaluated. Moreover, metadata server clusters for very large scale file systems present fundamental performance and reliability issues. For example, the metadata distribution produces situations where namespace operations such as `CREATE`, `DELETE` and `RENAME` of a file may require the activity of multiple MDSs to be performed in an atomic way.

Distributed namespace operations, also refered to as distributed transactions or simply transactions hereafter, represent a serious problem since they can produce file system inconsistencies if one of the involved MDSs fails before the transaction

978-0-7695-4844-9/12 $26.00 © 2012 IEEE
DOI 10.1109/ClusterW.2012.16

IEEE computer society

is completed. To overcome this problem, Atomic Commitment Protocols (ACPs) are used. This family of protocols guarantees transaction atomicity and then avoids file system inconsistencies even in the presence of failing servers. Indeed, either all the operations in the transaction are performed or none of them is performed at all.

One of the most popular and widely used ACPs is the Two Phase Commitment protocol (2PC) [5]. At the end of a distributed transaction, the protocol is started to make all the updates permanent in stable storage. Even if the 2PC protocol is widely used in transaction processing, it introduces a big overhead since it requires the exchange of a large number of messages between participants in the transaction, as well as expensive synchronous log writes that can seriously impact file system performance. For this reason several optimizations of the protocol, or even completely new solutions, have been proposed during recent years. These optimizations try to reduce both the number of messages and the synchronous log writes the protocol requires to commit a transaction.

In this paper we present a One Phase Commitment protocol (1PC) designed to reduce the 2PC overhead for all distributed namespace operations that, in a file system with distributed metadata service, require the activity of only two MDSs to be performed. In fact, `CREATE` and `DELETE` operations can involve up to two metadata servers in the cluster, whereas the `RENAME` operation can require up to four MDSs. Nevertheless, the number of `RENAME` operations in applications for High End Computing (HEC) clusters is negligible, whereas the number of `CREATE` and `DELETE` can become extremely high.

We developed a 1PC prototype using ACID Sim tools [6], an OMNET++ framework that provides a set of tools to write and debug ACPs. Using ACID Sim tools we also prove the protocol's effectiveness by comparing it with the 2PC protocol and two 2PC's optimizations called Presume Commit (PrC) [7] and Early Prepare (EP) [8]. We demonstrate that our protocol can increase the number of distributed create operations per second by more than 50%. This is achieved by eliminating the number of extra messages required to commit the transaction and by reducing the number of synchronous log writes. The proposed protocol can be particularly useful for those applications that require creation and/or deletion of a high number of files per second in the same directory [9] since it can reduce the number of synchronous log writes and, therefore, the contention on the directory. Furthermore, even if it is possible to adopt a metadata distribution strategy that preserves locality, limiting the number of distributed transactions, a very large number of concurrent operations in the same directory, which is a common use case, may turn the corresponding MDS into a bottleneck. It therefore makes sense to spread the files within the directory across multiple MDSs and use the proposed protocol to handle distributed transactions.

The remainder of this paper is organized as follows. In section II we describe the problem of managing distributed namespace operation and available ACP protocols to handle such operations. Section III describes the protocol we have developed for both failure-free and failure cases together with the assumptions we made in order to guarantee recoverability in case of failure. Section IV analyzes the protocol performance. Finally Section V and VI present related works on the problem of handling distributed namespace operations in Distributed File Systems (DFSs), conclusions and future works.

II. Motivation and Background

The number of clients in HPC clusters has increased constantly during the last years and will further increase in the future with the advent of the exascale computing age. In order to efficiently provide data to those clusters, file systems have evolved from centralized architectures, where all the data and the metadata were managed by a single server, to more complex architectures, where both data and metadata are distributed across many servers. The list of file systems that follow this approach includes Ceph [2] from the University of California Santa Cruz, FhGFS [3] from the Fraunhofer Competence Center for High Performance Computing and PVFS2 [4], just to mention a few. The Lustre community with the project Lepus [10] is also working on a distributed namespace for the Lustre file system.

Clustered metadata server architectures have become popular since they allow us to overcome the scalability issues related to having a single point of management for file system metadata. On the other hand the metadata distribution introduces a number of new problems. Here we focus on the problem of managing distributed namespace operations. Figure 1 depicts a simple example of a distributed namespace in a cluster of four MDSs. The figure shows that in some occasions a file and the corresponding parent directory can be assigned to different servers, e.g., in the case of *file1* and directory *dir2*.

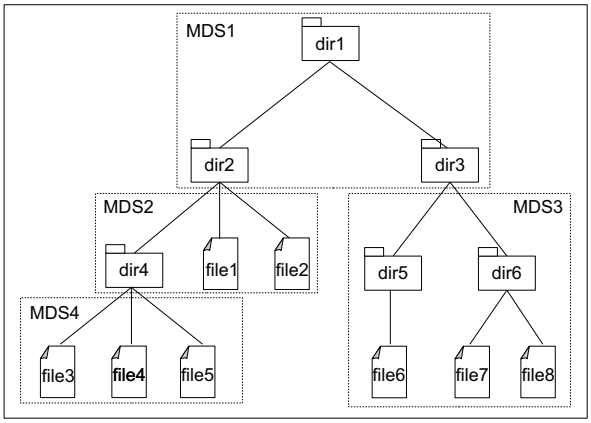

Fig. 1. Distributed Namespace Example: Every metadata server is responsible for providing clients with access to a piece of metadata information. Different dashed boxes in the figure correspond to different collections of metadata objects such as files and directories.

For this reason in clustered metadata server architectures, namespace operations such as `CREATE`, `DELETE` and `RENAME` may require the activity of multiple MDSs to be performed. To delete the file1 in Figure 1, for example, the

file system has to perform two different steps: (a) unlink the file's inode from the parent directory on MDS1 and (b) update the inode's reference counter and optionally delete the inode on MDS2. The following scenarios may occur when one of the involved MDSs fails during the execution of those steps:

- MDS2 deletes the file's inode but MDS1 fails before unlinking it from the parent directory. In this case the file has been deleted but the reference is still present in the parent directory.
- MDS1 unlinks the file from the parent directory but MDS2 fails before deleting the file's inode. In this case the partially completed operation leads to a situation where the file is still present in the file system but it is not referenced. Such a file is called an orphaned inode since it does not have a reference in the parent directory anymore.

All these cases produce inconsistencies since they violate file system invariants. The violated invariants for the two previous scenarios are: (a) if there is a name that references to a file, then that file exists, and (b) if a file exists, it is referenced at least once in the namespace.

The managing of distributed namespace operations in DFSs falls into the problem of transaction processing. A transaction is an indivisible sequence of computational steps, where every step can retrieve, modify and store data in the system. In order to be considered a Transaction Processing System (TPS), a DFS must satisfy the ACID properties. ACID literally stands for *Atomicity*, *Consistency*, *Isolation* and *Durability*. Atomicity, also known as all-or-nothing property, guarantees that all the computational steps in the transaction are performed completely, or none of them is performed at all; Consistency guarantees that the local invariants are preserved at every site; Isolation guarantees that concurrent transactions do not interfere with each other and, finally, Durability guarantees that once a transaction has been successfully completed all its results are durable even in the presence of failure.

To guarantee consistency, every node in the file systems forces metadata updates to a log in stable storage before reflecting them in the file system data structures. This mechanism is known as Write-Ahead-Logging and also ensures that logged updates are durable through system failures. In order to provide atomicity, DFSs usually adopt ACPs. The most popular and widely used between those protocols is the 2PC.

A. Two Phase Commit Protocol

There are several versions of the 2PC protocol that have been proposed and most of them try to reduce the protocol cost by making speculations about the outcome of the transaction. In order to make a distinction between all those versions, the baseline 2PC protocol is also called Presume Nothing (PrN) to remark the fact that no speculations on the transaction outcome are made. For this reason we will use the acronyms 2PC and PrN interchangeably.

Here we describe the protocol considering only transactions that require the cooperation of two MDSs. Figure 2 depicts the protocol behavior in the normal case. The 2PC is started at the end of a transaction in order to safely commit all the metadata updates to stable storage. The transaction starts when a client submits a request to an MDS to perform a distributed namespace operation (such as `CREATE`, `DELETE` or `RENAME` of a file). In order to keep track of the progress and handle failure cases, the 2PC also requires the receiving MDS, called *coordinator*, to assign an unique ID to the transaction and to write a `STARTED` record in the log. Afterwards the *coordinator* and the other MDS, called *worker*, perform their local updates in the cache, and when both have finished, the *coordinator* enters the 2PC protocol to safely commit them to stable storage. In order to accomplish this task, the protocol provides the execution of two separate phases:

1) during the first phase, also called *voting phase*:
   - the *coordinator* asks the *worker* to prepare to commit by sending it a `PREPARE` request. Furthermore, since the *coordinator* itself performs part of the updates, it also starts preparing.
   - the *worker* receives the `PREPARE` request from the *coordinator* and starts preparing. To prepare, the *worker* forces all the metadata updates from the cache to the log (Write-Ahead-Logging) and also writes a prepared record to remember its decision.
2) during the second phase, also called *commit phase*:
   - the *coordinator* receives the `PREPARED` message from the *worker*, meaning that it has logged its local updates and is ready to commit. At this point both the *coordinator* and the *worker* have prepared. The *coordinator* then writes a `COMMITTED` record in the log and sends a `COMMIT` request to the *worker*.
   - the *worker* receives the `COMMIT` request, writes a `COMMITTED` record in the log and replies with an `ACKNOWLEDGE` message. Afterwards, the log can be checkpointed and garbage collected.
   - the *coordinator* receives the `ACKNOWLEDGE` message from the *worker* and finalizes the log writing an `ENDED` record. At this point the transaction is committed and the log can be checkpointed and garbage collected.

As described above, the two MDSs can commit (as well as abort) a transaction by sending four messages and forcing metadata updates and protocol state information to stable storage during every phase. Thanks to the two separate phases and the synchronous write of the logs to stable storage, the 2PC protocol provides atomicity property to the transaction.

B. Two Phase Commit Concurrency Control

In order to provide isolation between concurrent transactions, the 2PC adopts a Two Phase Locking (2PL) mechanism. The 2PL mechanism ensures that the protocol acquires a lock for every metadata object that will be updated during the transaction. The locks are acquired at the beginning of the transaction before starting any update and are released only after the `COMMITTED` record has been written in the log. Furthermore, to avoid dead locks, the *coordinator* uses a timeout that allows it to abort the transaction releasing all the

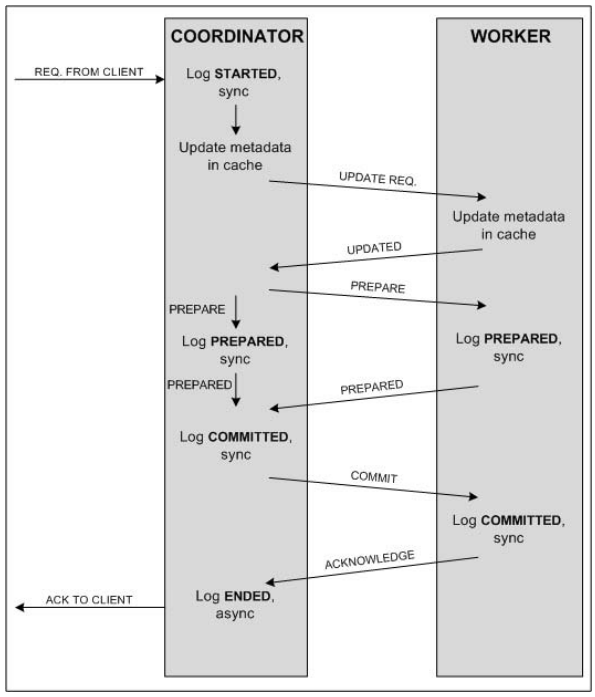

Fig. 2. The PrN protocol adds two message round trips and four write delays to the distributed namespace operation.

previously acquired locks if the *worker* does not reply to the `UPDATE REQ` within the predefined interval of time.

C. Two Phase Commit Failure Handling

To handle failures, the 2PC does not rely on any external failure detection system. The recovery protocol uses the information present in the log in order to commit or abort the transaction. If the *coordinator* crashes, it enters the recovery protocol upon restart and checks the log to find out the state of the transaction at the time the failure happened. The log in this case can contain the following records:

- **`STARTED`**: The failure occured before the *coordinator* has been able to prepare. In this case the transaction must be aborted since all the metadata updates have been lost in the crash. To abort, the *coordinator* sends an `ABORT` request to the *worker* and waits for the `ACKNOWLEDGE`.
- **`PREPARED`**: The failure occurred after the *coordinator* has prepared. In this case the *coordinator* resubmits the `PREPARE` request to the *worker* and continues with the normal protocol execution.
- **`COMMITTED`**: The failure occurred after the *coordinator* has received the `PREPARED` message from the *worker* but before receiving the `ACKNOWLEDGE`. The *coordinator* in this case resends the `COMMIT` request and continues with the normal protocol execution.
- **`ENDED`**: means that the failure occurred after the *coordinator* has received the `ACKNOWLEDGE` message. The transaction then is already committed and the *coordinator* takes no action.

In case the *worker* crashes, it checks the log upon restart to find out if there is some pending transaction and its state at the time the failure happened. The log can contain the following records:

- **`PREPARED`**: The failure occurred before the *worker* has been able to receive the transaction outcome from the *coordinator*. In this case the *worker* asks the *coordinator* to resend the decision and continues with the normal protocol execution.
- **`COMMITTED`**: The failure occurred after the *worker* has received the decision. In this case the *worker* takes no action.
- **`no entry in the log`**: If the *worker* receives a `PREPARE` request from the *coordinator*, since no entry is present in the log, it replies with a `NOT-PREPARED` message and takes no action. This happens when the *worker* reboots before preparing. On the other hand, if the *worker* receives a `COMMIT` request from the *coordinator*, this means that it had committed previously and the log entry is not present because it has already been checkpointed and garbage collected. This may happen if the *coordinator* does not receive the `ACKNOWLEDGE` message because of an unexpected reboot.

Failures not only involve MDSs but may also involve network components such as links between nodes. If a link fails, for example, the *coordinator* will not be able to communicate with the *worker* anymore. To avoid the *coordinator* to block for an undefined amount of time waiting for the *worker's* reply, the 2PC uses timeouts. If the *coordinator* does not receive the `PREPARED` message from the *worker* before a predefined timeout expires, it aborts the transaction. In this case when the connection is recovered, the *worker* will be informed of the abort outcome. The timeout mechanism also allows the *coordinator* to abort the transaction if the *worker* is too slow responding, thus limiting the duration of the transaction and reducing the contention on the metadata objects that can then be made available for other requests.

As described, the 2PC can guarantee ACID properties at the cost of large overheads introduced in the transaction. To reduce this cost, several optimizations have been proposed during the time. Here we discuss the PrC and the EP optimizations since they will also be used as a comparison metric for the evaluation of our protocol performance.

D. Presume Commit Optimization

The PrC optimization (Figure 3) saves one message and one forced log write compared to the PrN. This is accomplished by eliminating the `ACKNOWLEDGE` message. The aim of this message in the PrN is to allow the *coordinator* to finalize the log that afterwards can be checkpointed and garbage collected. Since a failing *worker* may ask the *coordinator* about the outcome of the transaction upon restart, the log should not be garbage collected until the *worker* has committed. On the other hand, the PrC optimization allows the *coordinator* to finalize the log after the commit outcome has been decided.

In this case if the *worker* asks for the transaction outcome and the log is not present at the *coordinator* site, the *worker* will presume that the outcome was a commit. In the abort case the PrC behaves in the same way as the PrN, meaning that all the messages and synchronous log writes are restored.

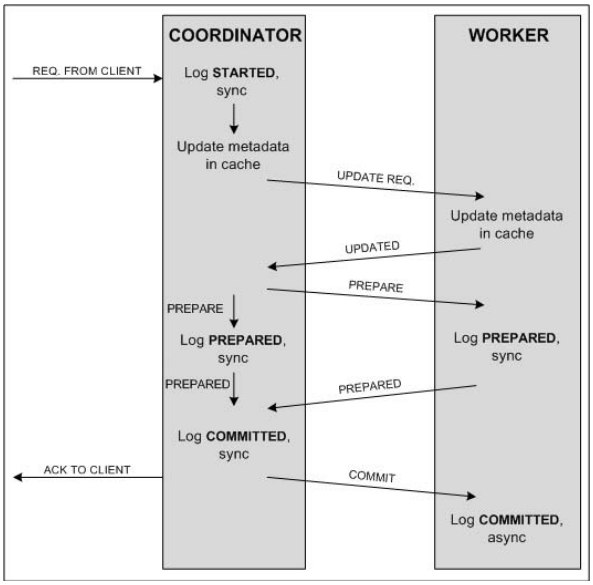

Fig. 3. The PrC optimization eliminates the acknowledge message and allows the *coordinator* to return to the client before the *worker* commits.

Even if the PrC can save one message and one forced log write, the *coordinator* still needs two messages for the voting phase (in order to collect the `PREPARED` from the *worker*), one for the commit phase (to forward the `COMMIT`) and wait for three forced log writes to stable storage in order to commit a transaction.

E. Early Prepare Optimization

A further reduction of the number of messages is provided by the EP optimization (Figure 4). In the EP protocol the *worker* autonomously prepares as soon as the last metadata update has been completed. This is done by piggybacking the transaction execution in the *voting* phase of the PrC protocol. Compared to this version the presented optimization can save two more messages. In terms of the number of sent messages and write delays, the EP protocol outperforms both the PrN and the PrC versions. Nevertheless, we think that there is still room for optimizations. Indeed, as already mentioned in previous sections, the basic assumption for this work is that namespace operations such as `CREATE` and `DELETE` involve only two MDSs in the cluster; therefore, since the commit decision can be reached involving the exchange of only one message between *coordinator* and *worker*, we can use one phase protocols.

III. One Phase Commit Protocol

In section II we have introduced the 2PC protocol and described the ideas behind the PrC and the EP optimizations.

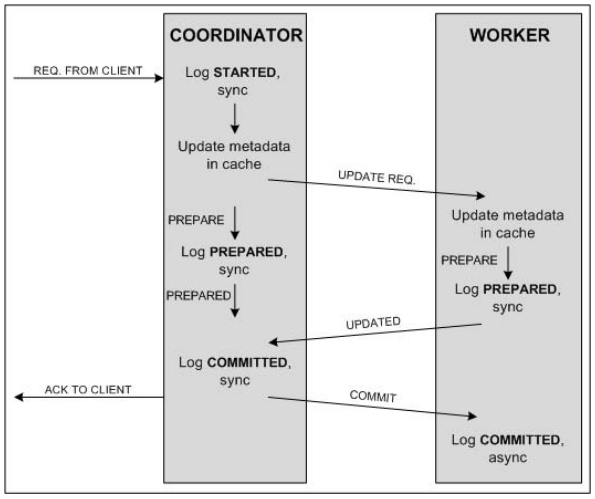

Fig. 4. The Early Prepare optimization piggybacks the prepare message in the job request, saving two messages.

We have emphasized that the 2PC protocol can guarantee transaction atomicity by running in two separate phases (*voting* phase and *commit* phase) and by logging state information as well as metadata updates to stable storage. In this work we focused on reducing the number of these phases from two to one, thus minimizing the communication overhead introduced by the 2PC protocol and the number of forced log writes. This result can be achieved by cutting off the voting phase and by piggybacking the transaction execution onto the commit phase. Indeed, the aim of the voting phase is to collect the `PREPARED` messages by the *coordinator* from all the involved *workers* and to allow the *coordinator* to make a decision about the transaction outcome. Since we are only interested in transactions that involve two MDSs, the *coordinator* in our protocol does not need to collect any `PREPARED` messages, instead it can ask the *worker* if it can commit and then decide accordingly.

Based on the described ideas, we have developed a tailored One Phase Commit (1PC) protocol to handle distributed transactions using only one phase while still guaranteeing atomicity. To achieve this, we have designed a completely new recovery mechanism, which assumes that the *coordinator* is able to access the log records of the failing *worker*. This assumption is legitimated by the fact that in clustered metadata server architectures metadata information must be made highly available in order to allow every MDS in the cluster to access it. This directly reflects on the architecture of the metadata servers cluster since it requires all MDSs to keep metadata as well as their logs in a central repository available to every other MDS.

A. Central Storage Architecture

In the 1PC protocol we assume that every MDS in the cluster keeps its log in a centralized storage device reachable via a Storage Area Network (SAN), for example. Every MDS maintains its log in a separate partition within the storage

device and every partition can be mounted and read by any MDS in the cluster. This assumption is made in order to implement a failure recovery mechanism that is able to satisfy the atomicity requirements of the protocol. The recovery is started by the *coordinator* every time it detects a failure of the *worker* involved in the transaction or when it reboots after a crash. The centralized storage architecture in this context provides a central repository that contains all the information about every distributed transaction that is running in the cluster. Such information can be used by the *coordinator* to find out the *worker's* decision whenever a problem occurs. This architecture represents the key point that allows the 1PC to correctly handle all the possible failure cases.

The centralized storage device just described is a shared resource and for this reason we have to make sure that it will be accessed in an exclusive fashion by all the MDSs. Indeed, as already mentioned, every log is written by only one MDS but can be read by anyone else in the cluster. This means that we have to guarantee that an MDS that is reading somebody else's log will be the only one accessing that log. In this case the MDS that owns the log must not be able to write it since it is supposed to have crashed. If not correctly handled, this scenario might cause file system inconsistency in the presence of a Byzantine failure. As an example, let us consider the case of a network partition between the *coordinator* and the *worker*. The failure detection system adopted in computer clusters to detect failing nodes is usually based on the exchange of heart beat messages. If a node does not receive heart beats from another node for a long period of time it declares that node as crashed. In the case of a network partition, the *coordinator* may wrongly detect the *worker* as crashed and start the recovery procedure by reading its log while it is still writing it, potentially producing a wrong transaction outcome (split brain problem).

Shared resources can be managed using fencing. Fencing is a mechanism that guarantees exclusive access to shared resources in a cluster, in this case the shared log on the shared storage device. There are two different types of fencing, resource fencing and node fencing. Linux clusters use a node fencing mechanism known as STONITH [11], literally *Shoot The Other Node In The Head*. The STONITH mechanism allows the alive nodes in the cluster to cut off the power supply of a node suspected to have failed, thus, forcing it to reboot. GPFS [12], for example, uses a similar approach to allow a designated node to recover the log of a crashed node on its behalf. The GPFS fencing mechanism is called disk leasing.

The resource fencing mechanism uses available resources to protect the shared storage device. In the case of a device accessible via a SAN network, the *coordinator* could isolate a *worker* beyond a network partition by instructing the fiber channel switch to reject all the requests coming from that specific node. The fiber channel switch in this case behaves as a barrier between the cluster nodes and the shared storage device. A further mechanisms for resource fencing is provided by the SCSI-3 standard through *Persistent Reservation* [13] that allows a SCSI drive to maintain a list of initiators that can access the drive.

Here we assume that the DFS can use one of the described mechanisms to ensure exclusive access to the shared logs. The identification of the specific mechanism to adopt is beyond the scope of this paper.

B. Failure-free Protocol

The protocol behavior in the normal case, where no failures occur, is reported in Figure 5. The transaction execution in the 1PC is piggybacked onto the commit phase using the same approach exploited by the EP optimization previously described.

The transaction is started when the *coordinator* receives a request from the client. The *coordinator* assigns a unique ID to the transaction and adds a record to the log to track its progress and recover in case of failures. This corresponds with the action of the 2PC *coordinator* at the beginning of a transaction. Besides the previously described protocols, in this case the *coordinator* also adds a redo record for the requested name space operation ("`CREATE` *filename*", for example). When the transaction information has been logged, the MDSs start with the transaction execution:

- the *coordinator* performs the first metadata update, sends an `UPDATE REQ` to the *worker* and waits for the `UPDATED` message.
- the *worker* receives the `UPDATE REQ` from the *coordinator*, performs the required metadata updates and starts the commit. To commit, the *worker* forces all updates from the cache to the log and writes a `COMMITTED` record. Afterwards, the *worker* replies to the *coordinator* with an `UPDATED` message.
- the *coordinator* receives the `UPDATED` message, replies to the client and starts the commit. To commit, the *coordinator* writes its metadata updates to the log and also writes a `COMMITTED` record. After the commit has been completed, the *coordinator* sends an `ACKNOWLEDGE` message to the *worker* in order to allow it to finalize its log.
- the *worker* receives the `ACKNOWLEDGE` message and finalizes its log writing an `ENDED` record. At this point the log can be checkpointed and garbage collected.

The commit of the transaction on the *coordinator* site starts asynchronously from the point of view of the client. This is possible since the *coordinator* writes a redo record in the log at the beginning of the transaction. The redo record can be used by the *coordinator* during the recovery protocol to re-execute all the metadata updates.

C. Failure Protocol

As already mentioned the 1PC adopts an aggressive recovery approach. If the *coordinator* crashes before the commit, it re-executes the transaction upon restart by accessing the information present in the redo record in the log. Furthermore, the *coordinator* starts the recovery protocol every time it is not able to get a response from the *worker*. From the recovery protocol's point of view there may, therefore, only be two failure cases.

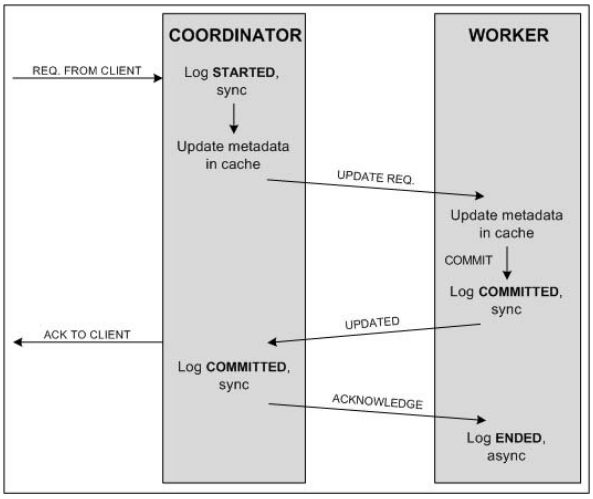

Fig. 5. One phase commit protocol failure-free behavior.

In the first case the *coordinator* checks its log upon restart to retrieve information about not completed transactions. The log in this case can contain the following records:

- **STARTED**: The transaction is not committed yet. Moreover, the *coordinator* does not know the state of the *worker*. In this situation the *coordinator* restarts the transaction from the beginning by making its local updates and re-submitting the `UPDATE REQ` to the *worker*. Then it waits for the `UPDATED` message.
- **COMMITTED**: The transaction is already committed and the *coordinator* does nothing.

A number of causes can be responsible for the second case. For example, the *coordinator* will not be able to get a response either if the *worker* has crashed or if a network partition has occured. Since the *coordinator* uses heart beat messages and timeouts to determine whether or not the *worker* has crashed, it will not be able to distinguish between the two cases. For this reason it cannot safely access the *worker* log to find out its decision. In this scenario, as previously described, the *coordinator* executes a fencing mechanism first to protect the shared log. For example, it can use the STONITH mechanism to force the *worker* to reboot and afterwards safely read its log. In the *worker* log it can find the following records:

- **COMMITTED**: The *worker* was successful in completing its updates and committing them. In this case the *coordinator* can commit.
- **no entry in the log**: The *worker* has crashed before committing its updates to the log. In this case, the *coordinator* has to abort.

Finally we analyze what happens on the *worker* site when it crashes. If the *worker* crashes, upon restart it checks its log to retrieve information about outstanding transactions. The log can contain the records:

- **COMMITTED**: The transaction has been committed but the *coordinator* may still need to access the log in the future. In order to finalize the log, the *worker* asks the *coordinator* to resend the `ACKNOWLEDGE` message.
- **ENDED**: The *coordinator* has committed and it does not need the log anymore. In this case the *worker* takes no action.

D. Concurrency Control

To provide isolation between concurrent transactions, the 1PC adopts the same two phase locking mechanism used in the 2PC protocol. What is different is that the lock at the *coordinator* site is released as soon as it receives the `UPDATED` message from the *worker*. In this case, no matter what will happen, the transaction will be committed eventually. Therefore, the *coordinator* can commit its updates to the log after it has sent the response to the client and released the locks on metadata objects. To avoid inconsistencies after a reboot, the coordinator will not execute new requests from any client until it has completed all the outstanding ones in the same order it received them before the failure happened.

IV. Simulation Framework and Results

In order to compare the performance of the presented protocols, let us consider Table I. For every protocol the table reports the total number of synchronous and asynchronous log writes, the number of synchronous and asynchronous log writes in the critical path, the total number of messages and the number of messages in the critical path.

The total number of messages and the messages in the critical path in Table I list the additional messages required by the specific protocol when compared with the case where no atomic commitment protocols are used. For instance, the PrN protocol requires four additional messages to perform a distributed namespace operation, whereas the 1PC requires only one more message (the `ACKNOWLEDGE` message). Therefore, the 1PC introduces no additional messages in the critical path.

TABLE I

	Total Log Write (sync, async)	Log Write in Critical Path (sync, async)	Total Messages	Messages in Critical Path
PrN	(5, 1)	(4, 1)	4	4
PrC	(4, 1)	(3, 0)	3	2
EP	(4, 1)	(3, 0)	1	0
1PC	(3, 1)	(2, 0)	1	0

Particularly relevant for application performance are the number of synchronous log writes and the number of messages in the critical path. These figures describe how long the *coordinator* takes before returning to the client with the transaction result. Moreover, since every metadata object has to be locked by the protocol just before starting the updates, they also determine when these locks will be released making

the metadata objects available for a new request. Usually the *coordinator* releases the locks after the `COMMITTED` record has been written in the log, that is concurrent transactions must not see partial results of each other in order to avoid unpredictable behaviors. Nevertheless, the 1PC *coordinator* can release the locks after the *worker* commits since at this time the *coordinator* is sure that all its local updates will be eventually committed to stable storage.

In order to evaluate our ideas, we have implemented the protocol using the ACID Sim Tools simulation framework. The simulator allows the user to define the architecture of the transaction processing system by specifying the number and the connection between a set of basic modules as well as the parameters for each of them. The basic modules are:

- `acp server`: Represents the computational node that executes transactions and runs the ACP protocol. This module can be connected to others of the same type to form a cluster of acp servers. Moreover, it is also connected to a `lock manager`, a `log manager`, an arbitrary number of `sources`, the `statistics` and the `leave` modules.
- `log manager`: Represents the stable storage where all the updates and state records are written. In the case of the 1PC, this module is connected to every `acp server` to implement the shared log architecture.
- `lock manager`: Manages the locks required by a transaction. There is a lock manager connected to every `acp server`.
- `source`: Represents the client that generates the transactions. Every source is connected to one `acp server` and to the `leave`.
- `leave`: Collects all the completed transactions in the system. All the aborted transactions can be resubmitted to the responsible `source` that reprocesses them.
- `statistics`: Collects statistics from acp servers.

Every acp server in the simulator can manage a user defined number of objects. From our point of view, these objects may be directories and files. For every object the acp server provides a list of methods that can be used to read or modify the object itself. These methods can be combined to build up transactions that are submitted to the acp server by the source modules. Furthermore, for every object the user can specify the size that object will occupy in the log as well as the time consumed by every method of that object. Finally the user can define the latency of the read and write operations from and to the log as inverse of the desired disk bandwidth[1] and the network latency for the exchange of messages between acp servers during a distributed transaction.

To run our tests, we have used the following parameters. We have considered a computational latency for every object, both for the read and the write methods, of $1\mu s$, a network latency of $100\mu s$ and a disk bandwith of $400KB/s$. Then, we have measured the number of distributed transactions per second that the system can sustain, using the previously described protocols. To do this, we have generated a synthetic workload where 100 distributed transactions are submitted at the same time to the same acp server. This workload intends to reproduce the behavior of HPC applications that create many files in the same directory. The results are reported in Figure 6. The best performance of all protocols is achieved

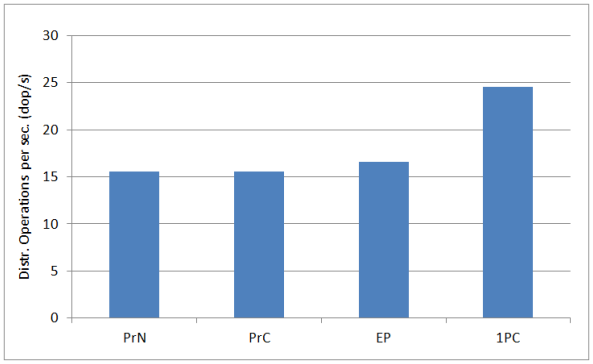

Fig. 6. Distributed Namespace Operations per second.

by the 1PC with 24 distributed transactions per second. PrN and PrC have nearly the same performance with 15 distributed transactions per second, while the EP performs slightly better with 16 distributed transactions per second. In conclusion, the 1PC protocol can gain more than 55% performance, compared with the PrN, whereas the PrC and the EP can gain only 0.39% and 6.60%, respectively.

V. Related Work

There is a number of research works on protocols to handle distributed namespace operations in DFSs. Most of them try to reduce the cost of the 2PC, minimizing the number of messages exchanged between MDSs as well as the number of forced log writes. Fan, Xiong and Ma [14] proposed a modification of the 2PC protocol for the Dawning Cluster File System that reduces the number of messages and forced log writes. The proposed protocol is based on the EP optimization of the 2PC protocol described in section II. Aguilera, Merchant, Shah, Veitch and Karamanolis [15] use the EP optimization as building block for the Sinfonia data sharing framework. The idea is to arrange operations in the transaction so that it is possible to piggyback the last operation execution onto the 2PC's voting phase, thus saving a message round-trip (minitransactions). Sinfonia also introduces a non-blocking failure recovery protocol for the *coordinator*. Ceph exploits metadata locality to reduce the number of distributed transactions and uses the 2PC protocol to guarantee consistency in the rare event that a distributed transaction occurs. Other projects try to reduce 2PC overhead by relaxing the ACID requirements for atomicity. Zhang and Karamanolis proposed an alternative approach for the DiFFS file system that can reduce the possible failures to non-severe inconsistencies such as orphaned inodes by re-ordering operations in the transaction

[1]This does not include separate contributions for seeking and rotational latencies but only the total latency required to write a block of data. This value has been chosen considering that shared storage access patterns can be highly random.

[16]. The ordered operations execution protocol adopts an aggressive recovery approach that requires the introduction of additional data structures, such as back pointers, in order to guarantee exactly one semantic. A similar idea is also adopted by Devulapalli and Wyckoff [17] who applied it to the PVFS2 file system. Finally Sinnamohideen, Sambasivan, Hendricks, Liu and Ganger [18] proposed a completely different approach for the Ursa Minor storage system. Here, all distributed metadata operations are reduced to local metadata operations by moving metadata responsibility from multiple MDSs to only one MDS that performs all the updates locally. The metadata migration approach is more heavyweight compared to the protocols discussed here since all the metadata objects must be moved between MDSs before they can perform any operation. Even if the performance penalty can be considered acceptable for `RENAME` operations that are very rare in HPC workloads, it becomes impractical for applications that performe a large number of `CREATE` and/or `DELETE` operations per second in the same directory, where the namespace is partitioned to multiple metadata servers.

In this paper we proposed a protocol that is different from any other protocol discussed in this section. Our protocol does not require to move metadata objects across MDSs in the cluster. Instead, we exploit the idea of shared storage architecture for both metadata and log information. Due to this, we can reduce the number of phases in the 2PC protocol by replacing the requirement of the voting phase with a rich and highly available source of information about every transaction running in the cluster.

VI. Conclusion and Future Work

The problem of managing distributed namespace operations in DFS can become especially serious for those HPC applications that generate a large number of `CREATE` or `DELETE` operations per second in the same directory. This is due to the need of guaranteeing the Isolation property that requires the parent directory to be locked before performing any operation. Furthermore, the locking of the parent directory is also required to meet the classical POSIX file system access semantic that guarantees a consistent view of the parent directory across multiple clients. The adoption of ACPs in this context has a negative impact since it introduces a big communication and log write overhead between the lock and unlock of the directory. In this paper we focused on reducing this overhead for a specific class of distributed namespace operations, which require the cooperation of only two metadata servers. The simulation results demonstrate that our protocol provides improvements by more than 55%. We think that these results are a good indication of how the protocol can perform in a real system, so the next step will be the implementation in a real distributed file system. Moreover, we think that further improvements in terms of the number of distributed operations per second may be achieved by aggregating multiple operations together. In this case we do not intend to change the semantics, instead we think that the MDS responsible for managing the parent directory can aggregate multiple namespace operations in only one big transaction, thus reducing the number of messages and log writes per block of requests. This is possible since the directory updates are all performed at the same MDS that can lock the object once and interleave expensive log writes with many operations in order to reduce the impact of the protocol on the performance.

Acknowledgement

This work has been supported by the Marie Curie Initial Training Networks (MCITN) of the European Commission (contract no. 238808), and the German Federal Ministry of Economics and Technology (BMWi) in the Simba project (grant KF21599005KM1).

References

[1] P. J. Braam, "Lustre: a scalable high-performance file system," White Paper, November 2002.

[2] 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 *Proc. of the 7th Conference on Operating Systems Design and Implementation (OSDI)*, 2006.

[3] FhGFS. [Online]. Available: http://www.fhgfs.com/cms/

[4] PVFS2. [Online]. Available: http://www.pvfs.org/

[5] J. Gray and L. Lamport, "Consensus on transaction commit," Microsoft Research, Tech. Rep., 2004.

[6] A. Mentis, P. Katsaros, and L. Angelis, "ACID Sim Tools: a simulation framework for distributed transaction processing architectures," in *Proc. of the 1st ACM International Conference on Simulation tools and techniques for communications, networks and systems (Simutools)*, 2008.

[7] B. Lampson and D. Lomet, "A new presumed commit optimization for two phase commit," in *Proc. of the International Conference on Very Large Data Bases*, 1993.

[8] J. W. Stamos and F. Cristian, "A low-cost atomic commit protocol," in *Proc. of the 9th IEEE Symposium on Reliable Distributed Systems (SRDS)*, 1990.

[9] E. Sminri, R. A. Aydt, A. A. Chien, and D. A. Reed, "I/O requirements of scientific applications: an evolutionary view," in *Proc. of the 5th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, August 1996, pp. 49–59.

[10] LEPUS. [Online]. Available: http://wiki.whamcloud.com/display/PUB/Remote+Directories+Solution+Architecture

[11] A. Robertson, "Resource fencing using STONITH," White Paper, August 2001.

[12] F. Schmuck and R. Haskin, "GPFS: a shared-disk file system for large computing clusters," in *Proc. of the 1st USENIX Conference on File and Storage Technologies (FAST)*, 2007.

[13] SCSI-3 Persistent Reservation. [Online]. Available: http://linux.die.net/man/8/fence_scsi

[14] Z. Fan, J. Xiong, and J. Ma, "A failure recovery mechanism for distributed metadata servers in DCFS2," in *Proc. of the 7th IEEE International Conference on High Performance Computing and Grid in Asia Pacific Region (HPCASIA)*, 2006.

[15] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis, "Sinfonia: a new paradigm for building scalable distributed systems," in *Proc. of the 21st ACM SIGOPS European Workshop (SIGOPS)*, 2007.

[16] Z. Zhang and C. Karamanolis, "Designing a robust namespace for distributed file services," in *Proc. of the 20th IEEE Symposium on Reliable Distributed Systems (SRDS)*, 2001.

[17] A. Devulapalli and P. Wyckoff, "File creation strategies in a distributed metadata file system," in *Proc. of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2007.

[18] S. Sinnamohideen, R. R. Sambasivan, J. Hendricks, L. Liu, and G. R. Ganger, "A transparently-scalable metadata service for the ursa minor storage system," in *Proc. of the USENIX '08 Annual Technical Conference (ATC)*, 2010.