Using Lightweight Transactions and Snapshots for Fault-Tolerant Services Based on Shared Storage Bricks

Michail D. Flouris[†]
ICS-FORTH
P.O. Box 1385, Heraklion
GR-71110, Greece
flouris@ics.forth.gr

Renaud Lachaize
ICS-FORTH
P.O. Box 1385, Heraklion
GR-71110, Greece
rlachaiz@ics.forth.gr

Angelos Bilas[‡]
ICS-FORTH
P.O. Box 1385, Heraklion
GR-71110, Greece
bilas@ics.forth.gr

Abstract

To satisfy current and future application needs in a cost effective manner, storage systems are evolving from monolithic disk arrays to networked *storage architectures based on commodity components. So far, this architectural transition has mostly been envisioned as a way to scale capacity and performance. In this work we examine how the block-level interface exported by such networked storage systems can be extended to deal with reliability. Our goals are: (a) At the design level, to examine how strong reliability semantics can be offered at the block level; (b) At the implementation level, to examine the mechanisms required and how they may be provided in a modular and configurable manner.*

We first discuss how transactional-type semantics may be offered at the block level. We present a system design that uses the concept of atomic update intervals *combined with existing, block-level locking and snapshot mechanisms, in contrast to the more common journaling techniques. We discuss in detail the design of the associated mechanisms and the trade-offs and challenges when dividing the required functionality between the file-system and the block-level storage. Our approach is based on a unified and thus, non-redundant set of mechanisms for providing reliability both at the block and file level. Our design and implementation effectively provide a tunable, lightweight transactions mechanism to higher system and application layers. Finally, we describe how the associated protocols can be implemented in a modular way in a prototype storage system we are currently building. As our system is currently being implemented, we do not present performance results.*

[†] Also, with the Dept. of Computer Science, University of Toronto, 10 King's College Road, Toronto, Ontario M5S 3G4, Canada.

[‡] Also, with the Dept. of Computer Science, University of Crete, P.O. Box 2208, Heraklion, GR 71409, Greece.

1 Introduction

Over the past decade, several trends have significantly shaped the design of storage systems. First, storage area networks (SANs) [20] that allow many servers to share a set of disk arrays, have emerged as a popular solution to improve efficiency and manageability, yet with limited extensibility and increased costs. Second, there has been much investigation on the benefits of moving file-system, database, or even application-level processing closer to the data store, e.g. offloading some tasks within the storage devices themselves [1, 19]. . Third, to achieve cost efficiency, storage systems will be increasingly assembled from commodity components, such as workstations, SATA disks and Ethernet networks. Thus, we are in the middle of an evolution towards a new storage architecture (often referred to as "storage bricks") made of many decentralized commodity components with increased processing and communication capabilities [9].

This new architecture has potential for scaling both storage capacity and performance. However, its distributed nature and the use of commodity components pose significant challenges towards the high-level of dependability that every storage system must guarantee. Given that it is desirable for performance and scalability purposes to eliminate all centralization points and increase I/O request asynchrony, it becomes difficult to deal with failures and to always maintain a consistent state of the system.

Recent networked storage architectures usually depend on a distributed file-system for providing reliability and availability (even if the underlying block layer provides RAID-type fault-tolerance). Thus, the complexity of determining consistency and ensuring persistence is usually built in the file-system. The main reason for this is that, traditionally, file-systems have been responsible for providing all the required mechanisms for sharing large scale storage system among multiple applications, whereas the block-level sub-

system was limited merely to performing simple read/write I/O operations.

However, cluster file-systems tend to become (i) scalability bottlenecks and (ii) unmanageable and difficult to customize. For these reasons, recent efforts [16, 17, 19, 25] are investigating block-level mechanisms that will lead to better scalability both regarding capacity and performance. Thus, we believe that an important problem is to examine how the re-design of the file-system and block-level subsystem roles will affect system reliability and availability and what mechanisms and approaches are required to deal with system failures.

Shared virtual disks [21, 5, 11] have been proposed to simplify the design of cluster file-systems. These infrastructures handle most of the critical concerns in distributed storage systems, including fault-tolerance, dynamic addition or removal of physical resources, and sometimes (consistent) caching. Nonetheless, the existing prototypes are monolithic and do no provide advanced features such as (file-system level) snapshots.

Besides, although various modular frameworks have been proposed for cluster storage systems [10, 14, 4], little work has investigated how fault-tolerance protocols used at different levels of the I/O stack (file and object/block interfaces) could be integrated in order to achieve simple design and optimized performance.

In this paper we present our design for a highly-dependable networked block-level storage system. Our approach relies on novel mechanisms that essentially provide distributed consistent storage snapshots using a transactional-type abstraction at the block-level. The client-side file-system is only required to use this abstraction without providing any additional support. This approach has been designed in the context of an extensible storage system [17, 18]. For this reason, the design of our mechanisms is amenable to a modular implementation; The system is designed as a set of virtual devices (layers) integrated in a virtual hierarchy distributed over many storage bricks. The focus of this paper is the detailed discussion of the reliability mechanisms. As the system is currently under implementation, we only briefly discuss performance implications. However, we expect the system to perform well, given that the proposed architecture eliminates all centralization points between clients and servers and synchronization (locking) services are distributed and scale well.

The rest of this paper is organized as follows. Section 2 provides an overview of our assumptions and design. Section 3 describes our scheme for achieving reliability. Section 4 discusses the recovery process after a failure has occurred. Finally, Section 5 draws our conclusions.

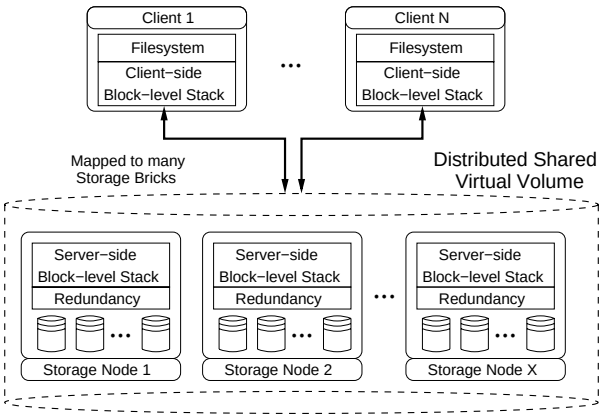

Figure 1. Structure of the system.

2 Overview

Although the flexibility of our system allows the construction of various kinds of storage systems, we are interested in networked storage systems that are shared at the block-level. Such a low-level, generic view of a storage container gives freedom to the client-side file-systems or other *block-level applications*[1] regarding the different abstractions that they can use. Next we present an overview of our block-level storage system.

2.1 System structure

This work is based on a *direct* storage topology, where each client has direct access to the set of storage nodes through a scalable network, such as Gigabit Ethernet or Myrinet. These storage nodes (also referred to as storage bricks or servers) have processing resources (CPUs and RAM) that are comparable to those of a workstation and contain a few tens of disks. This simple setup (Figure 1) is similar to a SAN. However, instead of custom networking fabric, such as FiberChannel, it uses a commodity network. Moreover, there is no central storage controller. Thus, it can be cost-effective while yielding good performance and scalability. On the other hand, reconfiguration in this setup cannot be made transparent to the clients because data placement/replication is handled on the client side. Note that our framework could allow to build other topologies that have been proposed[2] by various brick prototypes [3], but these developments are outside of our current scope.

[1] Note that since we present a block-level system, we consider a file-system running on top of it as an application. Another examples of block-level applications are raw I/O databases or web caches.

[2] The other common topologies are based on assigning one or several coordinators/gateways to each data object. Such a coordination role can either be mapped to an intermediate level of nodes or to the back-end storage bricks themselves.

Increased dependability and performance can be achieved through replication and striping both within and across bricks. The degree of replication, i.e. number of replicas, is configurable and independent of the reliability protocols we present. For simplification, we assume that our system uses a simple scheme for distributed redundancy (mirroring and striping) rather than more complex schemes, such as erasure coding, which are hard to implement in a decentralized system [13].

Finally, our design is fully symmetric: it does not require specific servers for metadata management and locking [12, 8], which may become a bottleneck. These services are instead provided in-band by the enhanced block layer. However, unlike traditional symmetric cluster file-systems [15, 7], client nodes in our system do not need to monitor each other to ensure progress despite failures, as in Frangipani [2].

2.2 Fault model

We assume that the networked storage system we are proposing behaves under the *timed asynchronous model* [6], which is among the most realistic ones for the system architectures we target.

We assume a fail-stop model for all the nodes of the cluster, i.e. clients and servers. Our approach handles any failure and network partitions. During failures, the availability of data is limited by their degree of replication. In other words, all non-faulty components of the system keep operating in the presence of faults, however data is only available if at least one copy is accessible through non-faulty components.

However, we do not deal with data corruption at the client side and/or incorrect client behavior. We consider these to be the responsibility of the (client-side) file-system. Moreover, we assume that the potential Byzantine behavior of the disks (or the low level device drivers of the OS) that may lead to data corruption is handled differently: the detection of inconsistencies, and possibly the recovery from such errors, are addressed with special modules deployed on each server node.

Finally, we assume clients belong to the same administrative domain as the storage bricks and do not exhibit Byzantine behavior. Nonetheless, we do not assume that clients have access to all data in the system but rather employ traditional permissions for access control.

Thus, in this work, we ensure that:

- Any transient or permanent (hardware or software) component failure should not corrupt or destroy stored information, unless it is the only remaining copy involved in the required operation.
- Any transient or permanent (hardware or software) component failure should not prevent the rest of the system from operating, unless it is the only remaining copy involved in the required operation.
- After any global failure, e.g. a total power failure, the system should be able to restart quickly from a consistent and recent state.

Overall, our approach deals will all these aspects of a dependable system, with the exclusion of handling Byzantine failures.

2.3 Application Guarantees

We have extended the traditional block-level API with new primitive operations. The features that our storage system provides to the file-system or applications through the block-level API can be described according to the common ACID classification:

Atomicity Using lightweight transactions, our system provides atomicity guarantees that are stronger than those provided by a single, local disk. A simple disk only enforces that a single sector is written atomically. We extend this guarantee to discontiguous ranges of sectors or blocks that may be mapped to multiple (and possibly replicated) servers. This is achieved by buffering the updates on the client-side until the transaction is closed.

Consistency A transaction is applied in a consistent way to all the involved servers. Thus, at any time, the replicas of a data object impacted by transactions will all be in the same state. This guarantee is obtained with a two-phase commit protocol.

Isolation This aspect refers to the serializability of transactions and is not provided directly by our transactional scheme, but through the use of locks. Our block-level API provides support for generic locks at the block-range granularity. Correct usage of locks is the responsibility of the application.

Durability Our basic transactions do not provide guarantees in terms of durability, meaning that updates committed in a transaction may be lost in case of a global crash before the transaction is committed in a snapshot. Our system, however, offers durability guarantees at the coarser granularity of snapshots (described in 3.3 and 3.4).

2.4 A new approach for dependable storage bricks

This originality of this approach lies in three main aspects: programming model, end-to-end management of de-

pendability issues and flexibility. First, we intend to provide simple abstractions for programmers that are developing dependable applications. Experience from file-systems and databases has shown that traditional techniques such as journaling are very complex to implement in a correct (and efficient) way. Instead, our proposition provides the programmers with a way to specify a set of atomic updates and their associated requirements in terms of durability. This model is simpler because it abstracts the application developers from low-level details such as journal management and write dependencies.

Second, we strive to address the inherent issues of a decentralized storage system in a unified way. Most layered systems use disjoint dependability protocols at different levels of interface. Typically, there is a multiple-phase protocol at the block level to enforce the consistency of distributed replicas [13, 25] and on top of this, a shared-disk file-system using another mechanism to ensure the atomicity of the updates (at least for metadata). The latter requirement is usually achieved with journaling [15], which involves synchronous writes and data copying. Unlike this traditional view, we consider that consistency and atomicity can be dealt with in a joint fashion, at a single level of the system. Our optimizations can be summarized as follows :

- It may not be necessary to enforce serializability of the updates at the level of the raw bricks, i.e. providing the same guarantees as a central storage controller, because application-level synchronization is sufficient[3].

- Atomicity semantics can be exposed to the application programmers without necessarily imposing durability. The loss of recent updates due to the rollback to the latest snapshot is a rare event that can only happen after a global system crash. In addition to durability, snapshots also provide backup versions of the data that can be used for disaster recovery, auditing, debugging, etc.

We expect this integrated approach to be more efficient that the traditional architectures described above since it involves simpler agreement protocols and fewer (less frequent) synchronous writes.

Finally, the few optimized distributed storage systems (such as GPFS [7]) addressing dependability issues in a single layer (usually at the file level), have a monolithic structure and can hardly be adapted or extended according to the needs of a given application. In contrast, the protocols that we propose can be easily added to (or removed from) a modular software stack for brick-based storage.

[3]We are interested in (distributed) applications with a pessimistic concurrency control scheme (typically file-systems with locking) but we think that this approach is also meaningful for applications relying on an optimistic concurrency control scheme (typically databases with timestamps).

3 Protocol Design

In this section we present in detail our design for ensuring that the system is always able to recover to a consistent point after a failure.

3.1 Lightweight Transactions

Maintaining data (metadata and user data) consistent in the presence of failures requires that we are able to:

1. Treat a sequence of operations (e.g. metadata updates for creating a file and inserting it into a directory) as an individual operation. That is, we need atomic updates.

2. Keep the distributed replicas consistent, if data is replicated on different storage servers. This is not trivial, because a client could fail in the middle of an update process, leaving some servers with a stale version of the data.

To address these two requirements we use a *lightweight* transaction mechanism, where metadata updates are protected against both kinds of threats. This is needed because the logical structure of the storage system (such as a file-system directory tree) must never become corrupted.

Transactions may also be used to guarantee the consistency of (user) data. However, a different trade-off may be adopted in this regard: sometimes, users are willing to accept occasional risks of data corruption[4] in exchange of increased performance. In this case, we can get rid of the first constraint (atomic treatment of multiple operations) and only enforce the second one (keep distributed replicas consistent despite failures). This is possible with a protocol based on *voting*: when a client reads a data item, the latter must make sure that a majority of replicas agree on the current value to be used. We believe that such a protocol should be faster than the one based on transactions for update-intensive workloads because the updates to the user data would not require a two-phase protocol[5]. The voting-based approach is nonetheless expected to have lower performance for read-mostly workloads because multiple servers must be contacted for every read operation.

The choice of the technique employed to keep the distributed data consistent only impacts two software layers: the application (e.g. the file-system) and the replication (e.g. RAID 1) module. We intend to base our first implementation on transactions for both metadata and user data. This decision was made both for simplicity (homogeneous usage of the same abstraction in the file-system,

[4]Because they can rely on backups or can easily regenerate the data (e.g. for a scientific calculation).

[5]And, as an additional optimization, a write request could be returned as soon as a majority of servers have acknowledged it (at the cost of decreased guarantees on the ability to recover up to the last written data).

lower development effort) and in oder to evaluate the performance penalty of the two-phase protocol for update-intensive workloads.

Next, we discuss three important aspects regarding transactions: their programming model, their semantics and their granularity.

3.1.1 Programming model

Our model of transactions is heavily influenced by the notion of "atomic recovery units" (ARUs) defined in the context of a centralized system, the Logical Disk by Grimm et al [22]. All the disk operations encapsulated in an ARU are guaranteed to be treated as a single operation during recovery (i.e. all or none of the operations are persistent after recovery). ARUs provide atomicity with respect to failures but not with respect to concurrency, where a complimentary serialization mechanism must be used for proper synchronization of different client threads.

A transaction is open and closed by the client application using explicit block-level API commands. For simplicity, we do not allow nested transactions in our programming model. Along with write operations, unlocks are also buffered at the client side and committed along with the transaction. This buffering is necessary, because unlocking a block range that was locked inside a transaction, before the transaction is committed, can break serialization semantics, since another client could complete another transaction dependent on this lock and commit it out-of-order. A transaction may contain several writes and lock/unlock operations.

Another important point is that locks handled by an application do not have to be systematically aligned with transactions. In our context, locks are conceptually used for *simple mutual exclusion*, i.e. to ensure that two sets of operations on a given resource do not overlap but are serialized. For instance, a client thread may lock a given file (i.e. a set of blocks)[6] to make sure that its updates will not be interleaved with updates from another client.

On the other hand, transactions are used for *atomic recovery*, that is to ensure that a given sequence of operations will be treated as a single operation during recovery (i.e. the whole set of operations will be considered as persistent or be cancelled).

It is important to note that while locking can be used to provide mutual exclusion independently of transactions, the opposite does not hold. In other words we cannot have atomic recovery (consistency) semantics without locks providing serializability. In this sense, locks used for atomic recovery can be mapped naturally to transactions but locks used to provide simple mutual exclusion should not. For instance, in a file-system:

- *i-node* locks are used for short updates to the metadata with critical requirements in terms of reliability. They should thus be associated with transactions.
- File locks granted to user applications are used for mutual exclusion and can be kept for a long time. As a consequence, it does not appear suitable to associate the acquisition (release) of a lock with the start (end) of a transaction. However, a set of atomic updates could be encapsulated in a transaction associated with a file lock. The distinction is that the lock could be held (much) earlier than the start of the first transaction and released (much) later than the end of the final transaction.

3.1.2 Granularity

Two levels of granularity can be considered for tagging transactions: node id or process/thread id. The node-level approach is easier to implement, requires less metadata for snapshots (see 3.3) and maps directly to the most common failure domain (assuming that hardware or OS crashes are more frequent than application-level crashes). On the other hand, the thread-level approach is conceptually clearer (there is no distinction between all the client threads, regardless of their home node), induces less contention on the client side for tagging the transactions (one counter per thread) and offers the potential to take snapshots more frequently (because of the finer granularity, more consistent points can be identified).

In the rest of the paper, we do not make any assumption regarding the chosen level of granularity and we just use the generic term "client stream".

The prototype that we are currently implementing works at the thread level granularity. This choice only impacts the client side module in charge of transactions. The rest of the code is generic in this regard.

3.2 Client-server Transaction Protocol

The transactional protocol involves two entities: the client-side transaction manager (CTM) deployed on all the client nodes and the server-side transaction manager (STM), deployed on all the storage bricks.

Once the end of a transaction has been detected, the CTM batches the data updates in a single `prepare` request, which is propagated to the STM modules through the hierarchy. At some point, this request is acknowledged and

[6] It may not be necessary to lock the whole range of data blocks to be updated, but only a few metadata blocks. For instance, a file-system could use a transaction to update both the metadata (inode) and the actual data of a given file. In this case, the file-system may be implemented in such a way that only the metadata blocks are locked (coarse grained locking at the file level). This is not a problem as long as all the file-system threads use the same techniques for synchronization.

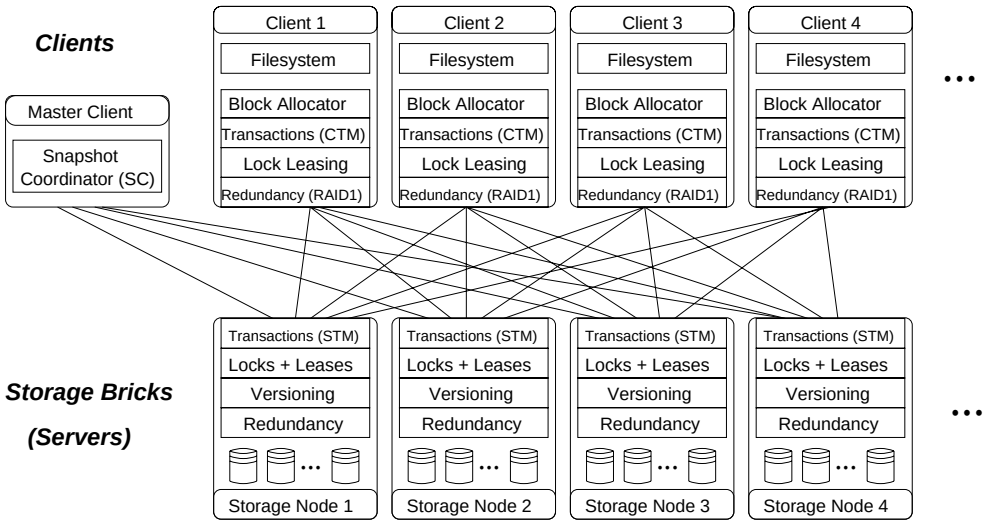

Figure 2. Fault-tolerant protocol stack.

the CTM checks its status. If the `prepare` request was successful, this means that all the concerned storage nodes have received the data to be written and agreed about it. Then, the CTM sends a `commit` request to all the involved storage nodes. Note that a `commit` request can be acknowledged before it has reached the disk. This is done on purpose, in order to lower the cost of the two-phase protocol, at the expense of weaker guarantees in terms of reliability.

On the STM, the committed write requests are placed in a queue, whose behavior depends on the current state of the snapshot protocol. When the snapshot protocol is not running, the queue is pass-through (it just logs the IDs of the transactions that are issued and keeps tracks of which ones have completed). When the snapshot protocol is running, the queue acts a buffer that allows to enforce a distributed agreement among the server nodes regarding the transactions that should be included in the next snapshot.

The STM layer only handles transactions and behaves as a pass-through layer for all other kinds of requests (including read, write, lock, unlock). As explained before, some setups may only use transactions for metadata updates and use basic write requests to access blocks associated with user data. Moreover, in both cases, reads are always issued as simple requests.

3.3 Snapshot Protocol

This protocol involves three main modules: the snapshot coordinator (SC) and, on the server side, the server transaction manager (STM, introduced in the previous section) and the version manager (in charge of the local snapshots).

To keep the description simple, we assume that there is only one snapshot coordinator, i.e. only one client node in charge of periodically triggering the protocol to create a new snapshot. However, in a realistic setup, this role may be attributed to several clients, for better load balancing and increased fault-tolerance.

The creation of a new snapshot relies on a two-phase protocol driven by the SC. The first phase aims at determining a globally consistent point for the snapshot. Upon reception of the message from the SC, the STM of each server temporarily queues the newly committed transactions and replies to the SC with a list specifying, for each client stream, the ID of the last transaction that was written to disk. The SC can subsequently examine the replies from all the servers and compute a globally consistent ”snapshot map”.

The second phase triggers the creation of local snapshots on the servers according to the snapshot map. Before a snapshot is actually taken on a server, transactions that are included in the snapshot map and not yet committed to disk are extracted from the queue and flushed to stable storage. The protocol used for the transactions guarantees that, for any transaction included in the snapshot map, each involved server has received the corresponding `prepare` request and thus, the required data updates.

The SC is also in charge of a periodic garbage collection protocol, aimed at reclaiming physical storage space. The latter essentially detects snapshots that are too old or that did not (globally) succeed and asks the bricks to destroy the corresponding version of their local volume.

3.4 Durability semantics

As previously described, the snapshot facility is based on the consistency intervals delimited by the lightweight transactions coupled with the use of locks. Different durability

semantics can be envisioned for the transactions, depending on the requirements of the (end user) application. On the one hand, it is possible to acknowledge a transaction (i.e. an atomic set of updates) as soon as all the servers have agreed on its status. In this case, if a global crash of the storage system occurs, it is not guaranteed that the recovery point will include the modifications even though the updates were successfully acknowledged to the applications. One could also envision a variant of the protocol where the storage nodes would not complete the commit protocol of a transaction before the related updates have been made persistent (i.e. after the next snapshot).

Our current prototype is based on the first strategy. However, the durability semantics could be configured in different ways: exposed as a volume-wide policy (for all the clients), tuned at fine grain (on a per-client basis), or exposed in the API for a single transaction. In addition, the API can also be extended so that client applications are (asynchronously) notified when a transaction is eventually synchronized to disk.

3.5 Protocol Implementation

We are currently implementing the protocols above within Locus [18], a low-level software framework aimed to take advantage of brick-based storage architectures. The modular nature of Locus allows tailoring the functionality provided by a clustered storage system to the diverse needs of applications (e.g. encryption, versioning, branching, caching, replication, duplicate elimination).

In addition, Locus supports sharing at the block level by providing (optional) locking and allocation facilities. Thus, such a configurable, block-level framework with an enhanced interface can significantly simplify the design and adaptation of complex software like cluster-file systems.

The fault-tolerance protocols are provided as a set of building blocks, which can be layered appropriately forming the desired stack of Locus modules. It is therefore possible to enable/disable them regardless of the ”functional” features of the storage system. However, the application must be modified to deal with the specific API for transactions. These changes will only impact a very restricted set of applications, such as a cluster file-system or a database management system, which may still export well-established, unmodified interfaces to end-user applications.

The core protocols are implemented with three Locus modules, two on the client side (snapshot coordinator and client transaction manager) and one on the server side (server transaction manager). The other functionalities, namely replication, leased locking, and (local) versioning, are provided by independent modules.

4 System Recovery

In this section we discuss how our approach recovers from each failure type. We discuss the three main types of failures: network partitions, client failures and storage brick (server) failures.

4.1 Network partitions

Temporary or permanent network partitions, although relatively unlikely, can put the system in a incorrect state and must be addressed. Several problems can arise because of a such a fault.

First, assume that we have two clients (C1 and C2) and two mirrored servers (S1 and S2). Because of a partition, we can end up with a “split brain” situation where C1 can only see S1 (and vice-versa) and C2 can only see S2. In such a configuration, the logical volume will “fork” in two versions. The same kind of problem can also happen if two servers disagree on the liveliness of a client that holds locks.

Second, even if all the servers agree on the fact that a client is dead, the latter may still be alive and believe that there is no problem. If the communication problem is only temporary then the client may think that it still holds locks (and an up-to-date cache), which may lead to corruption of the data and/or the file-system structure.

Overall, to operate correctly despite network partitions, the system must respect the two following invariants: (i) clients agree on the set of alive storage servers and (ii) storage servers agree on the set of alive clients.

The typical solution to deal with network partitions is to use a cluster manager. Thanks to heartbeat messages and voting, a quorum can be established among the cluster nodes[7]. Once a majority of the nodes agree on the current members of the cluster, the remaining nodes (considered faulty) must be isolated from their well-behaving peers through a fencing service. The fencing can be enforced at the hardware level (brutal power off via a remote power switch or network filtering thanks to a manageable switch fabric) or, in some cases, at the software level (through reconfiguration of the stack of the remaining nodes).

In our design, clients are not aware of each other and do not directly cooperate. Thus, one could believe that it is not necessary to include clients in the group monitored by the cluster manager and that using a lease-based protocol should be enough. However, the problem is actually more complex. For instance, the above mentioned “split brain” problem could not be detected with such a setup (because there is no mechanism allowing the servers to make an agreement on the current set of clients). A key difference

[7] Studies have shown that a cluster manager can be scalable provided that the underlying network features hardware support for multicast [24], which is the case for Ethernet.

with Petal/Frangipani in this regard is that, in Frangipani, a given lock is only managed by a single lock server, whereas in our design, we may have multiple servers in charge of the same (conceptual) lock (because we have one lock for each copy of a block). Thus, the complexity stems from the fact that we have multiple managers in charge of the same logical resource.

As a consequence, we take the following approach in order to respect the previously described invariants:

- We deploy a cluster manager (CM) only for the server nodes.
- The (distributed) CM does not only make decisions on the liveliness of the member nodes but also elects a leader among them. The leader server is assigned with an additional and specific network address, which is known by the clients.
- When the CM detects that a server node is not alive, it triggers a fencing procedure for the latter (see details below).
- On the clients, the management of network partitions only involves the communication layer.

When a client sends a request to a server, it arms a timer, which is destroyed when an acknowledgment is received. The expiration of a timer indicates that a server is unreachable. In this case, the client must contact the leader server to ask if the unresponsive node currently belongs to the group of alive servers[8].

If the leader replies that the server is not alive, then the communication layer returns an error to the upper layer[9].

If the leader replies that the server is alive, then this means that a network partition prevents the client from communicating with the server. In this case, the client should consider itself disconnected from the all the servers and the failed request should be acknowledged with a "disconnected" status. Upon propagation of the acknowledgment in the Locus hierarchy, all the modules will take the necessary measures to invalidate the state information that they hold (locks, cached data and metadata) . All the future requests should be rejected in the same way until a proper reconnection procedure succeeds.

If the leader does not reply, this means that a network partition prevents the client from communicating with some of the servers (or that all the servers are down). In this case, the disconnection procedure should be employed as well.

As explained above, both clients and servers may need to be fenced in order to preserve the integrity of a system experiencing network partitions. However, a distinction should be made on the severity of the fencing method to be used, according to the role played by a "failed" node. A server must be physically fenced because we need to make sure that all the clients always get the same view of the set of available servers. On the other hand, a client is responsible for fencing itself by voluntarily disconnecting from the set of servers. A self-fenced client does not have to be shut down or rebooted [10] and can (periodically) try to reconnect to the servers. Failure to reach all the alive servers (a list can be obtained from the leader) would indicate that communications issues are still occurring and that reconnection can not happen.

The above mentioned protocol related to network partitions is handled at the level of the communication layer. Thus, these issues do not impact the others parts of the I/O stack (the client-side modules only need to take the proper measures for state invalidation in case they receive a notification of disconnection).

4.2 Client failures

Throughout this section we assume that the client failures can be either due to a system crash (fail-stop model) or to network partitions (permanent or temporary loss of connectivity with the servers).

Locks: If a client fails while holding locks, this will eventually be detected by timeouts of the leases on a subset of the servers. On these nodes, the server leasing module will react to the timeout by reclaiming all the locks held by the client (sending `unlock` requests to the server-side locking module). If the client is still alive (i.e. the problem is not related to a crash but to communication issues), the previously described procedure for network partitions should be applied. If a client crashed without holding any locks, no recovery procedure is needed.

Transactions: The atomicity of a transaction is ensured by two properties: (i) the updates associated with a transaction are buffered on the client side until its closure and (ii) the 2-phase update protocol ensures that all the servers agree on whether a transaction should be committed or not.

If a client fails before a transaction is closed (or before any `prepare` request is sent), then the transaction will be automatically discarded because it will not reach any server.

If a client fails after sending the `prepare` requests (all of them or only a fraction of them), then the two-phase

[8] Note that the leader server should always be available. The use of a cluster manager for the server nodes ensures this invariant (if the leader crashes, a new one will be elected).

[9] Typically, upon notification of this error, an upper layer such as RAID 1 will switch to degraded mode and discard the path involving the failed server.

[10] However, the errors returned to the (user-level) application may require an application-specific treatment (possibly including a restart) but this is beyond the scope of our contribution.

protocol is not sufficient to make a decision (this is a well known issue of the 2PC protocol, which is blocking when the coordinator fails). To solve this problem (without assuming that the client may be able to come back quickly, if at all), we rely on server timeouts and the snapshot protocol. On each server, when the STM module receives a `prepare` request, it also triggers a corresponding timer. The timer is normally discarded when the associated `commit` request arrives. If the timer expires, the `prepare` request is considered as suspect and further inquiry will be necessary to determine if it deserves to be committed or not. The solution actually comes from the next round of the snapshot protocol: if at least one server has received a `commit` request, for the transaction, then the latter can be committed safely. Otherwise, the transaction should be discarded. This resolution process happens through the next "regular" round of the snapshot protocol. These principles also apply when the client fails while sending the `commit` requests (but in this case, we are sure that the transaction will be eventually committed).

Note that the protocol described above is not optimal because it may discard a transaction that could actually be committed (if all the concerned servers have received and acknowledged the `prepare` request and the client failed just before sending the first `commit` request). Yet, this scenario would seldom occur in practice and thus our approach trades robustness for simplicity in this regard.

4.3 Server failures

Server failures can either be related to physical disk failures or other software or hardware failures on the storage nodes.

4.3.1 Disk failures

The failure of a disk on a server can be masked to the client if enough redundant information has been stored on other drives (i.e. by means of a local RAID hierarchy) and if these drives are still available. In this case the server RAID layer must nonetheless notify the administrator about the failure. Once the disk is replaced[11], the administrator can issue a request to the server module to trigger the synchronization (a.k.a. "rebuild") of the new disk. Since we are in the context of a single node, the server RAID module acts as a central controller and classical rebuild techniques can be used.

If the disk failure cannot be masked by the server, the concerned request will fail, be returned to the client and reach the client RAID module (CRM), which replicates block to independent server nodes. For a read request, the CRM can then try to reissue the same request to another server node (which should hopefully succeed this time). A question is whether the node that returned the error should be discarded eagerly (i.e. as soon as a the problem is spotted) or if it should be kept online (because a subset of the data may still be available). The first option is simple but radical[12]. The second option is more complex and requires (i) to keep information on the client RAID module (in order to keep track of the blocks that are still valid on the deficient node) as well as (ii) to inform (through upcalls) all the clients that the server is in a degraded mode. For simplicity, we choose to discard a server node with deficient disk(s) in an eager fashion, similar to discarding a failed disk in a local RAID-1 module.

4.3.2 Node failures

Several situations can lead to the complete (fail-stop) crash of node. The root cause can obviously be a severe software problem (e.g. a kernel crash) or a hardware failure (e.g. a defective power supply). The stop of the node may also be triggered by a fencing operation due to (a) a network partition or Byzantine failure that made the node unresponsive to the cluster manager and led to its exclusion or (b) the detection of its disk failures.

In any case, once a server has been shut down, it cannot be later added to the pool of servers in a trivial manner (even if the hardware of software problems that led to its crash have been solved). Indeed the on-disk data and the (volatile) state of the Locus stack must be synchronized with the active servers. This process is not described in the present document due to lack of space.

5 Conclusions

Networked storage architectures based on commodity components promise more cost-effective and scalable storage systems both in terms of capacity and performance. However, they raise hard challenges in terms of reliability and availability, which translate into complex system design and overheads impacting system operation.

In this work, we discuss how reliability and availability can be provided in networked storage systems by using novel block-level mechanisms and exposing them through a well defined programming model (interface) to higher system layers, such as the file-system. This interface presents transactional features and our design aims at implementing these features with a low overhead at the block level.

[11] We assume that the machines support hot swapping for peripherals such as disk drives and NICs, which is nowadays a common feature. If this is not possible, the only way of increasing the robustness of the system is to add new server nodes. This second method requires support for dynamic reconfiguration [23], which is out of the scope of this paper.

[12] The fencing operation could be triggered by the defective server itself. The latter may just discard (return with an error) all the subsequent requests that it receives.

We detail the protocols that we use for achieving system consistency despite failures and we discuss the process for recovering from all major types of system faults (network partitions, crashed clients or storage nodes).

Finally, as our protocol is currently under implementation, we do not present any performance results. However, our design is such that (i) communication among clients is never necessary and (ii) clients are not required to synchronize during failure-free operation but rely on buffering in the storage nodes to ensure transactional behavior of I/O requests. Thus, we do not expect the I/O throughput to be affected significantly. However, we expect that our design may impact I/O response time when strict reliability (i.e durable commit semantics) is required, as requests have to be completed only after they have been made persistent, that is, after the next snapshot. It also remains to be seen whether the costs of the snapshot protocol (global agreement and synchronous writes for the metadata of the versioning layer) overcome those of (asynchronous) data copying required by a journaling approach.

6 Acknowledgments

We thankfully acknowledge the support of the European FP6-IST program through the UNIsIX project (MC-EXT-509595) and the HiPEAC Network of Excellence (NoE-004408).

References

[1] A. Acharya et al. Active Disks: Programming Model, Algorithms and Evaluation. In *Proc. of the 8th ACM ASPLOS Conference*, Oct. 1998.

[2] C. A. Thekkath et al. Frangipani: A Scalable Distributed File System. In *Proc. of the 16th ACM Symp. on Operating Systems Principles*, Oct.

[3] D. R. Kenchammana-Hosekote et al. The Design and Evaluation of Network RAID protocols. Technical Report RJ 10316 (A0403-006), IBM Research Almaden, 2004.

[4] E. Zadok et al. FiST: A Language for Stackable File Systems. In *Proc. of the 2000 USENIX Technical Conference*, June.

[5] E. K. L. et al. Petal: Distributed Virtual Disks. In *Proc. of the 7th ASPLOS Conference*, Oct. 1996.

[6] F. Cristian et al. The Timed Asynchronous Distributed System Model. *IEEE Transactions on Parallel and Distributed Systems*, 10(6), 1999.

[7] F. Schmuck et al. GPFS: A Shared-disk File System for Large Computing Centers. In *Proc. of the 1st USENIX FAST Conference*, Jan. 2002.

[8] G. A. Gibson et al. A Cost-Effective, High-Bandwidth Storage Architecture. In *Proc. of the 8th ACM ASPLOS Conference*, Oct. 1998.

[9] J. Gray. Storage Bricks Have Arrived. Invited Talk at the 1st USENIX FAST Conference, 2002.

[10] J. Hartman et al. The Swarm Scalable Storage System. In *Proc. of the 19th IEEE ICDCS Conference*, May 1999.

[11] J. MacCormick et al. Boxwood: Abstractions as the Foundation for Storage Infrastructure. In *Proc. of the 6th USENIX Symp. on Operating Systems Design and Implementation*, Dec. 2004.

[12] J. Menon et al. IBM Storage Tank - A heterogeneous scalable SAN file system. *IBM Systems Journal*, 42(2), 2003.

[13] K. A. Amiri et al. Highly Concurrent Shared Storage. In *Proceedings of* 20^{th} *IEEE ICDCS Conference)*, 2000.

[14] K. Amiri et al. Dynamic function placement for data-intensive cluster computing. In *Proceedings of the USENIX Technical Conference*, 2000.

[15] K. W. Preslan et al. Scalability and Recovery in a Linux Cluster File System. In *Proc. of the 4th Annual Linux Showcase and Conference*, 2000.

[16] M. Abd-El-Malek et al. Ursa Minor: Versatile Cluster-Based Storage. In *Proceedings of the 4th USENIX FAST Conference*, 2005.

[17] M. D. Flouris et al. Violin: A Framework for Extensible Block-level Storage. In *Proc. of 13th IEEE/NASA Goddard Conference on Mass Storage Systems and Technologies*.

[18] M. D. Flouris et al. Shared & Flexible Block I/O for Cluster-Based Storage. Technical Report 380, Institute of Computer Science, FORTH, Greece, July 2006.

[19] M. Mesnier et al. Object-Based Storage. *IEEE Communications Magazine*, 41(8), 2003.

[20] B. Phillips. Industry Trends: Have Storage Area Networks Come of Age? *Computer*, 31(7):10–12, July 1998.

[21] R. A. Shillner et al. Simplifying Distributed File Systems Using a Shared Logical Disk. Technical Report TR-524-96, Princeton University, 1996.

[22] R. Grimm et al. Atomic Recovery Units: Failure Atomicity For Logical Disks. In *Proc. of the 16th IEEE ICDCS Conference*, 1996.

[23] R. Lachaize et al. Simplifying Administration through Dynamic Reconfiguration in a Cooperative Cluster Storage System. In *Proc. of 5th IEEE Cluster Conference*, Sept 2004.

[24] W. Vogels et al. Scalability of the Microsoft Cluster Service. In *Proc. of the 2nd USENIX Windows NT Symposium*, 1998.

[25] Y. Saito et al. FAB: Building Distributed Enterprise Arrays from Commodity Components. In *Proc. of the 11th ASPLOS Conference*. ACM Press, October 2004.