RIBD: Recoverable Independent Block Devices

Name(s) omitted for blind review process

Abstract

Scalable storage systems built with collections of commodity components offer the promise of cost-efficient infrastructure for a variety of scalable applications. Such storage systems are often designed to implement persistent and highly-available storage abstractions such as virtual disks, exported through a standard block-level interface. Applications over these systems require additional transactional support to achieve atomic, consistent, and scalable access to their distributed data. As a result they tend to be complex and often to duplicate functionality already available in the underlying storage system.

In this paper we examine the alternative of a scalable storage system that exports a richer, transactional block-level interface to applications. Our system, named *RIBD*, relies on a type of lightweight transactions (termed *consistency intervals*) for expressing higher-level data consistency requirements and on *low-overhead versioning* at the block level for failure recovery. *RIBD* uses these block-level mechanisms for dealing with both data and metadata consistency, presenting a simple abstraction to higher system layers. Our contributions in this paper are: (a) the design and implementation of a *RIBD* system prototype and the underlying protocols; (b) an evaluation of the *RIBD* prototype and a comparison to two existing cluster file systems, GFS and PVFS2. Overall, we find that *RIBD* provides stronger consistency semantics while performing comparably to these file systems. *RIBD* achieves this through clean and simple abstractions, whose applicability extends beyond cluster file systems.

1 Introduction

Cluster-based storage architectures in use today resemble the structure shown in Figure 1-(a) where a high-level layer such as a file system uses the services of an underlying *block-level storage system* through a standard block-level interface such as SCSI. Concerns such as data availability and metadata recovery after failures are commonly addressed through data redundancy (e.g., consistent replication) at the file or block (or both) layers, and by metadata journaling techniques at the file layer. This structure, where both reduncancy and metadata consistency is built into the file system, leads to filesystems that are complex, hard to scale, debug and tune for specific application domains [1, 2, 3].

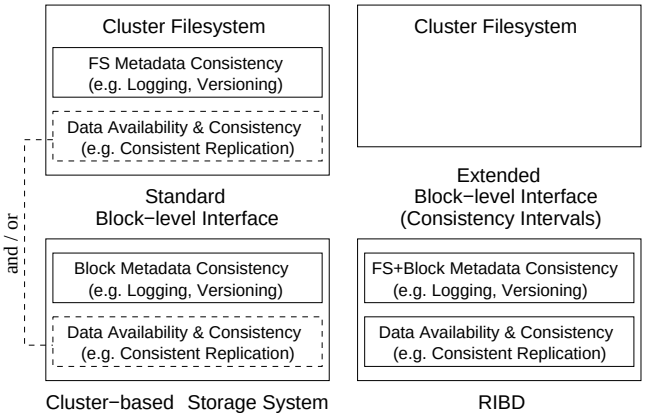


Figure 1: Overview of typical cluster-based scalable storage architectures and *RIBD* .

Modern high-end storage systems incorporate increasingly advanced features such as thin provisioning [4, 5], snapshots [6], volume management [7, 8, 9], data deduplication [10], block remapping [11, 12, 13, 14], and logging [13, 15]. The implementation of all such advanced features requires the use of significant block-level metadata, which are kept consistent using techniques similar to those used in filesystems (Figure 1-(a)). Filesystems running over such advanced systems often duplicate effort and complexity by focusing on file-level optimizations such as log-structured writes, or file defragmentation. Besides doubling the implementation and debugging effort, these file-level optimizations are usually irrelevant or adversely impact performance [16].

In this paper we propose *RIBD*, an alternative cluster-based storage system (depicted in Figure 1-(b)) that moves the necessary support for handling both filesystem and block-level metadata consistency to the block layer. To this end, *RIBD* needs to provide an enhanced block-level interface to the filesystem. *RIBD* proposes the notion of *consistency intervals* (CIs) to provide fine-grain consistency semantics on sequences of block-level operations by means of a lightweight transaction mechanism. *RIBD* extends the traditional block I/O API with commands to delineate CIs, offering a simple yet powerful interface to the file (or block-level application) layer.

Another distinctive feature of *RIBD* is its roll-back recovery mechanism based on low-cost versioning. We believe that versioning is a particularly promising approach for building reliable storage systems as it matches current

disk capacity/cost tradeoffs. To the best of our knowledge, *RIBD* is the first system to propose and demonstrate the use of versioning for recovery purposes at the block storage level.

We implemented *RIBD* in the Linux kernel as a kernel module exporting virtual devices layered over local or remote physical storage devices. We also implemented a simple file layer called *ZeroFS* (*0FS*) over *RIBD*. *0FS* is a *user-level*, *stateless*, *pass-through* file system that merely translates file names to sets of blocks and specifies CI boundaries in file operations. We evaluated *RIBD* using micro-benchmarks on a platform of 24 nodes (12 disk servers and 12 application/file servers). To understand the overheads in our approach, we compared *RIBD* with two popular file systems, PVFS [17] and GFS [18], which however offer weaker guarantees. We found that *RIBD* offers a simple and clean interface to higher system layers and performs comparable to existing solutions with weaker guarantees.

The rest of the paper is organized as follows. In Section 2 we discuss our motivation and related work and in Section 3 an overview of *RIBD*. Sections 4, and 5 present the design of the underlying protocols and components as well as the steps taken for recovering from faults. Sections 6 and 7 discuss our prototype implementation and the platform we use for our evaluation. Section 8 presents our experimental results. Finally, section 9 draws our conclusions and outlines future work.

2 Motivation and Related Work

RIBD is a distributed storage system that offers transactional guarantees to applications through a block-level interface to the underlying storage. Our decision to choose a block-level interface for *RIBD* rather than a higher-level (e.g., file) interface is based on a number of reasons. First, a block-level interface to storage can potentially be used by a wide variety of applications, whereas a higher-level interface to storage is confined to a more narrow set of applications written specifically to that interface. Second, storage systems exporting a block-level interface, such as logical volume managers and storage controllers, have in recent years grown in functionality and intelligence incorporating advanced support for scalability, availability, and disaster recovery. Often, system support for data and metadata consistency is already present within those systems. We believe that it is only natural to take advantage of existing support and extend it appropriately to offer applications the level of transactional guarantees that they need without the burden of having to (re-)implement it, duplicating functionality already available in other system layers. Finally, we believe that offering transactional guarantees to applications for enabling consistent access to their state (e.g., file system metadata) is easier performed at the block level. The alternative of performing it within the application (i.e., through the use of a journal [19, 20, 21]) and over a variety of possible implementations of block-level storage systems is fraught with challenges arising from the different specifications and guarantees that these implementations provide [22, 23]. We believe that a transactional block-level interface to storage such as that provided by *RIBD* addresses these concerns.

We also believe that a key component of any storage system is support for maintenance of previous versions of data in a structured manner, a feature known as *versioning* [24, 25], allowing recovery to *any* previous state of the system. Versioning has a number of important uses in system introspection through past states and data recovery after operator or system failures. In designing *RIBD* we incorporated versioning at the block-level within a distributed storage system. To the best of our knowledge *RIBD* is the first system using distributed block-level versioning for roll-back recovery to the most recent valid data. Our contributions include the design and implementation of distributed algorithms for its implementation within a scalable storage system.

A number of past research projects relate to *RIBD* in different aspects of its design. Storage systems that provide a transactional API at the block level include Echo [26], Base Storage Transactions (BSTs) [27], and Logical Disk [28]. Echo provides block-level transactions using a redo log for recovery. Unlike Echo, *RIBD* achieves recovery using distributed versioning techniques. BSTs [27] allow the system to maintain consistency at all block operations on distributed RAID arrays. The *RIBD* API uses the notion of *consistency intervals* (CIs) to achieve atomicity, consistency, and durability, relying for isolation to the use of explicit locks within CIs in a more general manner. Finally, unlike Echo and BSTs, our evaluation of *RIBD* is based on a real prototype. CIs are similar to atomic recovery units (ARUs) in Logical Disk [28]. However, unlike ARUs, CIs deal with replica consistency as they target distributed scalable storage systems.

Systems that use versioning at the file level include zFS [29], Elephant [24], 3DFS [30], the Inversion file system [31], Plan9 [32], and self-securing storage [33]. Also, Cedar [34] and Venti [10] use a similar concept of immutable files. Finally, the Comprehensive Versions File System (CVFS) [25] uses techniques to reduce metadata overhead in versioning file systems. Systems that use versioning at the block level include WAFL [35], and SnapMirror [36]. *RIBD* differs from them in its support for globally consistent block versions.

Similar to *RIBD*, systems such as the Hurricane file system [37] and FiST [38] have extended the functionality of the operating system I/O stack. However, these systems operate at the file level whereas *RIBD* appears as

a block device to the operating system.

Another system related to *RIBD* is Sinfonia [39], a system that supports mini-transactions with configurable durability semantics on memory servers. There are nonetheless significant differences between *RIBD* and Sinfonia. First, Sinfonia proposes a relatively general-purpose transactional abstraction while the *RIBD* model is simpler and focuses on block-level storage systems. In addition, Sinfonia follows a roll-forward log-based recovery model while *RIBD* implements a roll-backward versioning-based recovery scheme. In addition the cluster file system built on top of Sinfonia explores a different design space by supporting coherent, client-side caching, with a write-through read-validation strategy. Our approach would also impose a write-through scheme for (optional) client-side caching at the end of a transaction. However, the use of explicit locking would allow client caches to avoid read validation. Our approach also bears similarities with Swarm [40], an extensible storage service based on a low-level, striped log abtraction, but our designs differ in terms of programming interfaces and recovery techniques. Besides, to the best of our knowledge, the Swarm project did not study in depth features such as distributed atomic updates and data sharing.

RIBD is related to FAB [41] in its use of a voting protocol to deal with all types of failures in replica consistency. Unlike CIs, FAB allows updates to replicas to complete when quorum is guaranteed. During reads, a voting process takes place and decides the value of the block. This approach incurs fairly different tradeoffs compared to our protocols (writes are faster and reads are slower). Supporting a voting mechanism for replica consistency of non critical data in our system would only require modifications to a small subset of modules. However, we choose to use a single, transactional mechanism for all purposes, as we believe it is more appropriate for primary storage applications, as opposed to archival-type applications.

3 *RIBD* Overview

RIBD tries to address issues at three levels: (a) abstraction and primitives it presents to higher layers, (b) cost-effective scalability without compromising reliability, and (c) flexibility in its use. In this section we present an overview of *RIBD* and discuss the abstraction it provides. The next sections discuss primarily how it deals with consistency issues and to some extend how its implementation results in flexibility.

Previous work and other systems show that explicitly handling atomicity in higher layers, e.g. by using journaling alongside non-trivial file semantics, results in complex systems [1]. File systems that support journaling tend to be hard to implement and even harder to debug, modify and tune [2, 3]. Moreover, as lower system layers provide higher level abstractions of system resources, e.g. by abstracting block allocation and placement in a networked system, solutions for atomicity become difficult to optimize for performance. For these reasons, *RIBD* hides this complexity by providing to higher system layers and applications a simple abstraction based on *consistency intervals* (CIs), similar to transactions. Higher layers can delineate a set of consecutive (block) operations as a consistency unit that will be handled appropriately by *RIBD*. Figure 2 shows an example code segment from *0FS* that uses the *RIBD* API for the file creation operation.

```
...
begin_ci();
  lock( directory );
  if ( !lookup( filename, directory ) {/*file exists?*/
     allocate_inode( &inode_addr );
     lock( inode_addr );
     write_file_metadata( inode_addr, name, attrib );
     unlock( inode_addr );
     init_dirent( dirent, inode_addr, name, attrib );
     write_dir_metadata( directory, dirent );
  }
  unlock( directory );
end_ci();
...
```

Figure 2: Pseudo-code for a simple file create operation with a single CI.

To provide cost-effective scalability without compromising reliability, *RIBD* uses a *single, lightweight* transactional mechanism for both replica and metadata consistency, based on CI interface. Typically, file system approaches require different mechanisms for dealing with metadata consistency and replica consistency, as these refer to different entities, data blocks, file system metadata, and consistency metadata (e.g. the log itself). *RIBD*, by operating at the block level is able to use a single mechanism for all types of blocks, regardless of whether they refer to file data or metadata. Our transactional mechanism uses agreement protocols for consistency, low-overhead versioning for atomicity, and relies on explicit locking for isolation. Given that all requests are eventually written to disk, durability is provided in a straight-forward manner at a configurable granularity by specifying a flush interval. The combination of these low-level mechanisms results in little contention when there is no (true) access sharing at the file system, does not impose dependencies during system operation for concurrent I/Os, and only increases the response time of individual I/O requests, if the application requires the ability to recover after each, single I/O operation.

Finally, the few optimized distributed storage systems (such as GPFS [42]) addressing dependability issues in a single layer, the file system, have a monolithic structure and can hardly be adapted or extended according to the needs of a given application, e.g. by providing customizable replication. In contrast, the protocols that we propose can be easily added to (or removed from) a modular

software stack and operate at the block level.

3.1 System abstraction

Overall, CIs provide atomicity, consistency, and durability. Isolation is provided by a separate locking API. All block operations enclosed in a CI are guaranteed to be treated as a single operation during recovery, i.e. all or none of the operations persist after a failure. A CI is opened and closed by the client application using explicit block-level API commands. Given that we are operating on the critical I/O path, we do not allow nested CIs. Essentially, a CI buffers all updated operations on the client side, until the commit operation.

Higher system layers use *RIBD* CIs to delineate units of work that may leave the system in inconsistent state after a failure. This includes all critical updates to metadata operations, because the logical structure of the storage system, such as a file system directory tree, must never become corrupted. CIs may also be used to guarantee the consistency of data. However, a different trade-off may be adopted in this regard: sometimes, users may be willing to accept occasional risks of data corruption, e.g. if they can rely on backups or can easily regenerate the data, in exchange of increased performance. In this case, the higher system layer need not include accesses to data in CIs. This decision on how CIs are used is left to the file system. In our work and *0FS* we choose to ensure both metadata and data consistency, because managing inconsistencies with increasing volumes of information, quickly becomes a difficult problem.

As mentioned, *RIBD* CIs do not provide isolation. This is achieved with appropriate block-level locking operations in the file system code. Typically, such locking operations will occur within CIs. Thus, along with write operations, we also need to buffer unlock operations that incur within CI boundaries, to avoid cascading effects during recovery.

We choose to *not* implement locking transparently at the CI boundaries, to allow for possible optimizations at the file level, e.g. by using a different granularity for atomicity and for mutual exclusion. Locks are only required for mutual exclusion, that is 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 (or range of blocks) to ensure that its updates will not be interleaved with updates from other clients. However, in a file system, specific metadata maybe associated with specific data blocks. In this case, it may be adequate for the client to include all metadata and data operations in a transaction but only lock the metadata blocks.

In summary, *RIBD* provides atomicity, consistency, isolation, and durability properties for CIs as follows:

Atomicity Our mechanism guarantees that the updates encapsulated in a CI will be performed atomically, i.e. that a server will perform either all or none of the updates but never any other combination. This is achieved by buffering the updates on the client-side until the CI is closed.

Consistency CIs complete only after they are applied to all involved servers and replicas. Thus, at any time, all data replicas impacted by CIs will be in a consistent state. This is achieved by means of a two-phase commit protocol among disk servers.

Isolation CIs do not support implicit isolation. Instead, higher layers can guarantee isolation by using explicit block-range locks that are provided by *RIBD*.

Durability The updates encapsulated in a CI become durable once two conditions apply : they have been included in a globally consistent version and each disk server has flushed the corresponding write requests to its local disks. The durability semantics and the commit latency seen by the RIBD clients can be configured according to various trade-offs in terms of reliability and performance, as explained in section 4.2.

3.2 Fault model

We assume that failures of client and server components (CPUs, memories, disks, software faults) are fail-stop and the whole node containing the component crashes. Similarly, we assume that network components fail permanently, and that transient network failures at links or switches are dealt with by the communication subsystem, as is typical for high-throughput low-latency interconnects.

In case of any failure, data remains available as long as there is a functional path from the application to the specific data blocks. 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.

RIBD does not actively employ mechanisms to detect incorrect behavior. It assumes that component errors are reported through return codes of synchronous or asynchronous operations. Thus, *RIBD* does not deal with undetected errors or Byzantine behavior. Finally, we assume that file and disk servers belong to a single administrative domain and are co-located. Thus, we do not deal with disaster recovery issues.

4 Underlying Protocols

Figure 3 illustrates the system structure. Table 1 describes the base functionality of each module.

4.1 Client-server transactional protocol

The protocol implementing atomicity involves two entities: the client-side CI manager (CCM) deployed on all

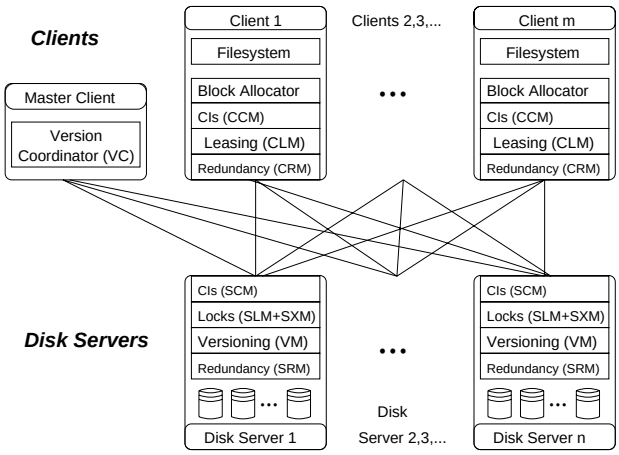


Figure 3: *RIBD* system architecture and protocol stack.

Client Modules		Server Modules	
CCM	CI Manager	SCM	CI Manager
VC	Version Coordinator	VM	Versioning Module
CRM	Redundancy Module	SRM	Redundancy Module
CLM	Leasing Module	SLM	Leasing Module
		SXM	Lock Manager

Table 1: Client (left) and server (right) module functions in *RIBD*. *Client* denotes block-level modules on file servers, whereas *Server* denotes block-level modules on disk servers.

the client nodes and the server-side CI manager (SCM), on all the disk servers, as shown in Figure 4.

CCM buffers the write and lock operations associated with a given CI (and services read requests from the buffer if necessary). Once the end of a CI has been detected, the CCM starts the two-phase commit protocol: First, it batches the data updates in a single `prepare` request, which is propagated to the SCM modules. At some point, this request is acknowledged by the SCM on the servers and CCM checks its status. If the `prepare` request was successful, this means that all the concerned disk servers have received the data to be written and agreed on it. Then, CCM sends a `commit` request to all the involved disk servers. 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 SCM, the committed write requests are placed in a queue, whose behavior depends on the current state of the versioning protocol. When the protocol is not running, the queue is pass-through (it just logs the IDs of the CIs that are issued and keeps tracks of which ones have completed). When the versioning protocol is running, the queue acts as a buffer that allows to enforce a distributed agreement among the server nodes regarding the CIs that should be included in the next version.

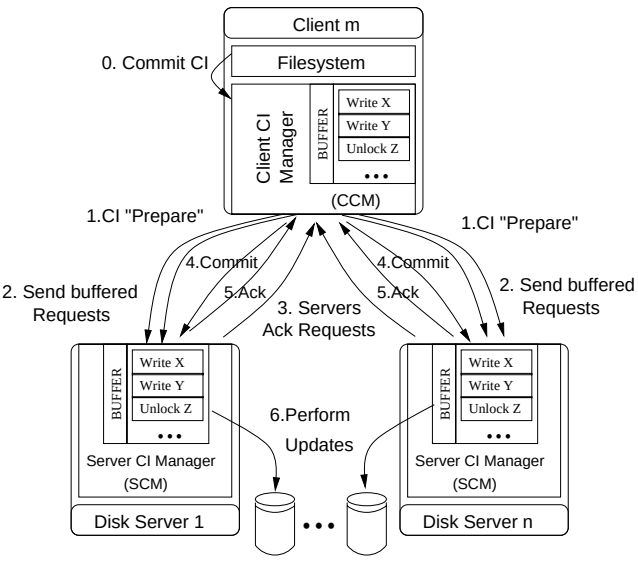


Figure 4: Transactional protocol between the client CI manager and multiple server CI managers.

The SCM only handles CIs and behaves as a pass-through layer for all kinds of requests that are outside the scope of CIs (including read, write, lock, unlock). As explained before, some setups may only use CIs for metadata updates and use basic write requests to access blocks associated with user data.

4.2 Versioning protocol

Versioning involves three main modules: the version coordinator (VC), the server CI manager (SCM), and the version module (VM) that in charge of local versions. To simplify our description, we assume that there is only one version coordinator, i.e. only one node in charge of periodically triggering the protocol to create a new version. However, in a realistic setup, this role may be attributed to several nodes, for better load balancing and increased fault-tolerance. Besides, applications may also interact with a VC module to request the creation of a new version. Figure 5 shows an overview of the the versioning protocol in *RIBD*.

The creation of a new version relies on a two-phase protocol driven by the VC. The first phase aims at determining a globally consistent point for the version. Upon reception of the message from the VC (step 1), the SCM of each server temporarily queues the newly committed CIs and replies to the VC with a list specifying, for each client stream, the ID of the last CI that was written to disk (step 2). The VC can subsequently examine the replies from all the servers and compute a globally consistent "version map", which is sent to the servers (step 3).

The second phase triggers the creation of local versions on the servers according to the version map (step 5 and 6). Before a version is actually taken on a server, CIs that

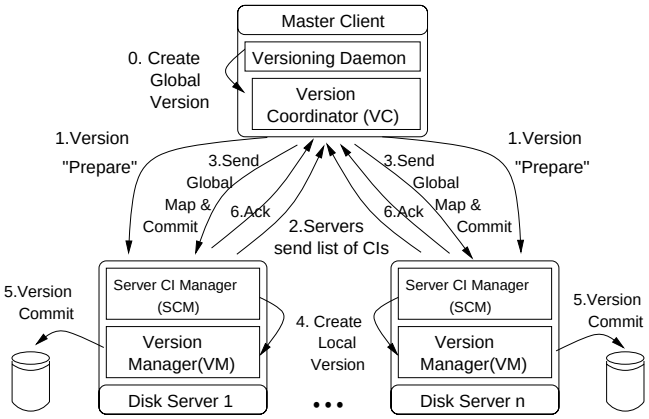


Figure 5: Versioning protocol between the versioning coordinator, all server CI managers, and server versioning modules.

are included in the version map and not yet committed to disk are extracted from the queue and flushed to stable storage. The protocol used for CIs guarantees that, for any CI included in the version map, each involved server has received the corresponding `prepare` request and thus, the required data updates.

After a global crash of the system (e.g. a power failure), VC coordinates the recovery process by communicating with the storage nodes in order to determine the most recent and complete version that can be used. This role is assigned to a single instance of the module.

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

We use versioning to offer different recovery guarantees, depending on the requirements of (end user) applications. First of all, the durability semantics can be configured on a per-volume basis. According to the needs of a given application, a CI can be acknowledged to a client at one of the two following stages: (i) as soon as all the servers have agreed on its status (but not performed the related I/O operations on disk) or (ii) only after the CI is persistent. In the first 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. The system can still recover to a previous consistent point, however, it will have acknowledged a CI that was eventually not committed. In the second case, all CIs committed are guaranteed to be part of the system after recovery.

Second, the CI commit latency is adjustable. In the extreme case, each CI commit can cause a new version. However, to improve the global performance of the system, it is more appropriate to batch many disk updates related to multiple CI commits.

4.3 Replication protocol

Availability can be achieved through redundancy mechanisms, such as replication or erasure coding. In our system we use data replication across different disk servers (RAIN: Redundant Array of Independent Nodes) as the only form of redundancy, rather than more complex formulas, such as erasure codes. The degree of replication (i.e. number of replicas) is configurable and independent of the reliability protocols we present. Currently, we only support RAID-0 and RAID-1 functionality.

4.4 Other protocols

Locking Mutual exclusion between clients for data access is achieved the use of locks. The locks are provided for block ranges and are handled by the disk servers. The information on the current state of the locks is not stored in stable storage but only in memory. Locks are associated with leases, renewed periodically by the clients. The locks held by a crashed client can thus be reclaimed.

Liveness 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, which uses heartbeat messages and voting, in order to establish a quorum among the cluster nodes. 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 protocol stack of the remaining nodes).

In our system, clients are not aware of each other and do not cooperate directly, which complicates handling "split brain" scenarios, because there is no mechanism allowing the servers to make an agreement on the current set of clients. We deal with this by deploying a (distributed) cluster manager (CM) only for the server nodes, which does not only make decisions on the liveness 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 it.

5 System Recovery

In this section we briefly discuss recovery from two main types of failures: network partitions and client (file server) failures. We omit details concerning local disk

and disk server failures since they are respectively handled using traditional RAID and distributed-RAID techniques.

5.1 Network failures

On the clients, the management of network partitions only involves the communication layer. When a the client does not receive a reply from a server within a given time interval, it must contact the leader server to ask if the unresponsive node currently belongs to the group of alive servers. There are three possibilities:

(i) The leader replies that the server is not alive, where the communication layer returns an error to the upper layer.

(ii) 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 considers 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 *RIBD* hierarchy, all the modules will take the necessary measures to invalidate the state information that they hold (locks, cached data and metadata). All future requests should be rejected in the same way until a proper reconnection procedure succeeds.

(iii) The leader does not reply, which means that a network partition prevents the client from communicating with some or all of the servers. In this case, the disconnection procedure is employed as well. Since the above protocol is handled at the communication layer, these issues do not impact the rest of the I/O stack (the client-side modules only need to ensure proper measures for state invalidation in case they receive a notification of disconnection).

5.2 File server/client failures

RIBD does not rely on client metadata for correct operation, and thus, client failures require little maintenance. The main issues are: releasing any acquired locks and guaranteeing appropriate operations of CIs.

First, as mentioned previously, if a client fails while holding locks, this will eventually be detected by timeouts of the leases on the servers. Second, the atomicity of a CI is ensured by two properties: (i) the updates associated with a CI 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 CI should be committed or not.

If a client fails before a CI is closed (or before any `prepare` request is sent), then the CI will be automatically discarded because it will not reach any server. If a client fails after sending the `prepare` requests (a fraction or all of them), then the two-phase protocol is not sufficient to decide. This is a well known issue of the 2PC protocol, which is blocking when the coordinator fails [43]. Our solution, without assuming that the client may be able to recover quickly, if at all, relies on server timeouts and the versioning protocol. On each server, when the SCM module receives a `prepare` request, it also arms 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 should be committed or not. The solution actually comes from the next round of the versioning protocol: if at least one server has received a `commit` request, for the CI, then the latter can be committed safely. Otherwise, the CI should be discarded. This resolution process happens through the next "regular" round of the versioning protocol.

Note that this protocol is not optimal because it may discard a CI that could actually be committed (if all the concerned servers have 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 recovery to a less recent point for simplicity in this regard.

6 System Implementation

We implement all related protocols under *ABC*, an extensible block-level framework for decentralized, cluster-based storage architectures. *ABC* is implemented as a block device driver module in the Linux 2.6 kernel accompanied by a set of user-level management tools. *ABC* supports sharing at the block level by providing (optional) locking and allocation facilities.

We implement all *RIBD* protocols related to reliability as a set of building blocks, which can be layered appropriately in a *ABC* virtual hierarchy, as shown in Figure 3 and Table 1. It is therefore, possible to enable/disable all reliability extensions, regardless of the "functional" features of the storage system, e.g. the specific type of replication provided. In total, *RIBD* consists of approximately 40K lines of kernel code and 15K lines of user-level code.

Next, we comment on implementation issues of each system component. CI management modules (CTM, STM) implement the core of the CI mechanism. The versioning module (VM) creates and manages remap-on-write versions of a local volume. It is placed right on top of the SRM so that the "real" data and the metadata from all the above modules can be treated in the same fashion). It interacts with the server CI manager (STM) in the context of the versioning protocol. Version coordination (VC) is a lightweight process and will typically be performed by a single VC that is elected once, at boot time. Another VC election can be triggered anytime the current VC fails and is removed from the system. Redun-

dancy modules (CRM, SRM) operate in the same way both for the disk and file server sides. Neither SRM nor CRM does need to run an agreement protocol. SRM handles only local replication, whereas for CRM agreement is handled by the two-phase protocol by the CTM. Also, the client leasing modules (CLM) works in conjunction with the server leasing module (SLM). These modules need to be "paired" in all *RIBD* configurations, where locking is necessary.

ZeroFS: To evaluate our approach we build ZeroFS (*0FS*), a *stateless, pass-through* file system that translates file calls to the underlying *RIBD* block device in the OS kernel. *0FS* allows shared, distributed access to files from different nodes. Unlike distributed file systems but similar to Frangipani [44], *0FS* does not require explicit communication between separate instances running on different application nodes. Typically, communication is required for agreement purposes. Instead, *0FS* uses the corresponding block-level mechanisms provided by *RIBD* volumes.

0FS is implemented as a user-level library that provides I/O operations on files and directories. Being user-level, *0FS* needs to cross the kernel boundary several times during a file operation, whereas a kernel implementation need only perform a single crossing.

0FS does not support a (client-side) cache but relies on *RIBD* for any caching it may perform. Our current design and implementation of *RIBD* does not support client-side caching. We believe that for scaling to large numbers of clients, client state should not affect system state required for recovery purposes. Thus, existing approaches for client-side caching need to be re-thought, especially given the availability of high-throughput low-latency interconnects, which is beyond the scope of our work. As shown in our results (Section 8), the use of a client-side cache does not improve, but rather degrades performance when the workload is not fs-metadata-intensive (i.e. consists of few large files and directories as our IOzone experiments). Such workloads are typical to many parallel applications.

7 Experimental Platform

Our evaluation platform is a 24-node cluster of commodity x86-based Linux systems. All cluster nodes are equipped with dual AMD Opteron 244 CPUs and 1 GByte of RAM, while 12 nodes acting as disk servers have additionally four 80GB Western Digital SATA disks each. All nodes are connected with a 1 Gbit/s Ethernet network (on-board Broadcom Tigon3 NIC) through a single 24-port GigE switch (D-Link DGS-1024T). All systems run Redhat Enterprise Linux 5.0 with the default 2.6.18-8.el5 kernel.

To examine the overhead of our protocols compared to existing systems we compare *RIBD* against two other systems: A cluster filesystem, Global File System (GFS)[45] system, and a parallel filesystem, Parallel Virtual File System (PVFS2) [17, 46]. These file systems offer weaker guarantees compared to *RIBD*. We choose to use them because (a) they are widely used in real setups and (b) contrasting *RIBD* to them will reveal the cost of offering stronger guarantees based on our approach.

In the GFS setup we use GNBD and CLVM provided by RedHat's Enterprise Linux 5.0. Note that neither GNBD nor CLVM support data replication. Thus, to configure a replicated (RAIN-10) setup for GFS we use the Linux software RAID driver (MD), which, however, does not maintain replica consistency and does not incur any related overheads. In this setup, GFS reliability guarantees are weaker than *0FS*. PVFS2 is only available with support for striping configurations because it uses its own networking protocol, which does not support data replication. In our setup we use the kernel module of PVFS2 for the clients and ext3 for the server-exported storage.

All three filesystems (GFS, PVFS2, and *0FS*) are installed and evaluated on the same 12 cluster nodes acting as clients and using the same 12 server nodes with 48 disks in total. In the case of GFS and PVFS2 we have tuned the filesystems for optimal performance according to the vendors manuals. In our evaluation we use two cluster I/O benchmarks: IOZone [47] and *clustered* PostMark [48].

IOZone is a benchmark that generates and measures a variety of file operations. We use the distributed IOZone setup in version 3.233 to study file I/O performance for the following workloads: sequential read and write, random read and write, reverse read and stride read. In all workloads we vary the block size between 32 KBytes and 16 MBytes. We use a different 8-GByte file for each client. The aggregate data volume for every IOZone run is 96 GBytes of data for all 12 clients.

PostMark [48] is a synthetic, filesystem benchmark that creates a pool of continually changing files on a filesystem and measures transaction rates for a workload simulating a large Internet electronic mail server. The original version of PostMark is a single-node application. To use it in our setup, we modify its initialization and termination code using MPI to: (a) spawn processes on a cluster of nodes, (b) synchronize the various benchmark phases, and (c) communicate aggregate results at the end. Note that the benchmark code itself remains unchanged. In our setup, each client node/process uses a different directory on the cluster filesystem. The transactions issued by each process consist of (i) a create or delete file operation and (ii) a read or append file operation. Each transaction type and its affected files are chosen randomly. When all transactions complete, the remaining files are deleted.

We use two workloads for PostMark (Table 2) that differ in file size distribution (1-10 MBytes for the first and

	File Sizes	Initial Files (per client)	Transactions (per client)	Files Created (aggregate)	Read traffic (aggregate)	Write traffic (aggregate)
Workload A	1 - 10 MB	800	4000	33588	162 GBytes	223 GBytes
Workload B	10 - 100 MB	100	300	3072	126 GBytes	191 GBytes

Table 2: Cluster PostMark workloads.

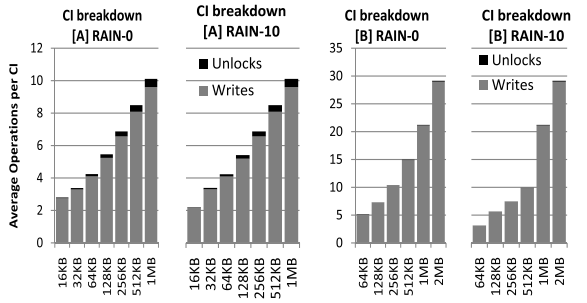
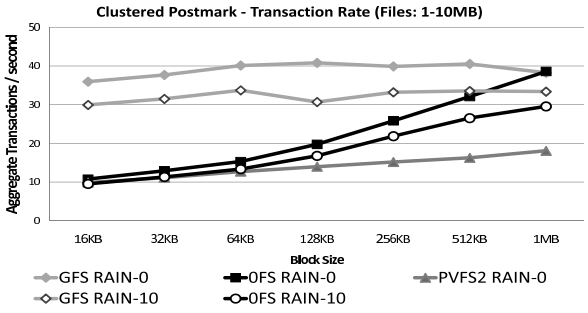
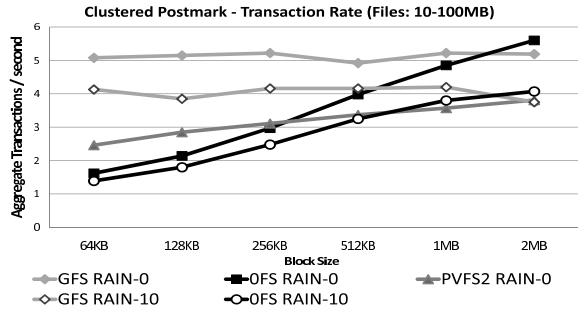


Figure 6: Average number of operations per CI and breakdown to write, unlock operations for Postmark.



(a) Workload A (1 - 10MB files).



(b) Workload B (10 - 100MB files).

Figure 8: PostMark aggregate transaction rate (transactions/sec) for workloads A and B.

10-100 MBytes for the second), the number of initial files per client, and the number of transactions per client. Both workloads create a sufficiently large number of files and enough read and write traffic to fill in the node caches and allow for consistent results.

To facilitate interpretation of results, we use a symmetric system configuration with 12 disk servers and 12 file servers/application clients for all filesystems. For the configuration of *0FS* we have used the protocol stack shown in Figure 3 and two data distribution setups: (i) a striped volume with no redundancy, where all 48 disks in the cluster are striped at the disk server side (RAID-0) and the disk servers are striped at the file server side (RAIN-0). (ii) a RAIN-10 volume with consistent RAIN-1 replication, where the disks are striped at the disk servers using RAID-0 and the disk servers are mirrored and striped at the file server side (RAIN-10). In all RAID and RAIN setups we use a chunk-size of 128 KBytes.

All data and metadata passing through *RIBD*'s stack, both in RAIN-0 and RAIN-10 are versioned by the server-side versioning modules. In Section 8.5 we study the overhead of the versioning agreement protocol, measuring 0FS with Postmark under four versioning frequencies. In the rest of the experiments, we do not capture versions during their duration.

Figure 6 shows the average number of write and unlock operations in each CI for workloads A, B, for different request sizes. Writes, refer to individual I/Os that are performed to disks by the disk servers. Although the number of operations specified by the application in a CI is statically known, the number of I/Os that are eventually generated by these operations depends on the block placement. Since we measure this statistic on the disk server side, RAIN replication does not affect the number of I/Os in each CI (but affects the number of CIs that disk servers see).

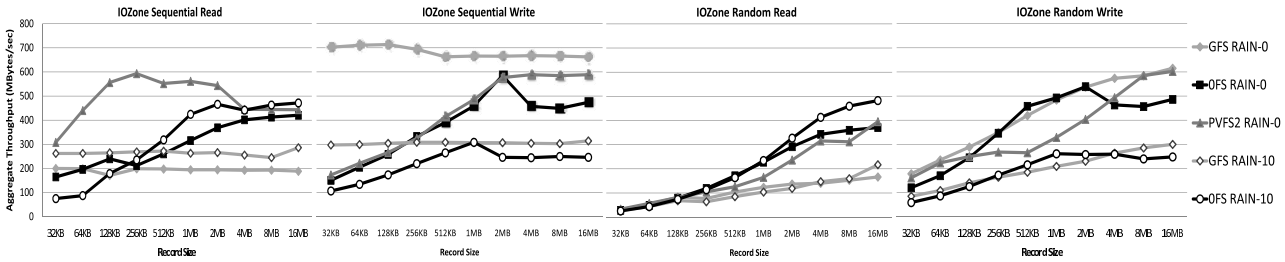
8 Experimental Results

This section examines the overheads associated with our approach on a setup with multiple storage and filesystem nodes.

8.1 Overhead of Dependability Protocols

Figure 7 shows IOZone results for sequential and random workloads. There are five curves on each graph, showing the three filesystem in a RAIN-0 setup and additionally GFS and *0FS* on a RAIN-10 configuration. As mentioned, GFS in the RAIN-10 configuration does not support a replica consistency scheme, and thus, has lower overhead.

In the case of sequential read (Figure 7-a) *0FS* outperforms GFS and is very close to PVFS2 for large block



(a) Sequential Read (b) Sequential Write (c) Random Read (d) Random Write

Figure 7: IOZone sequential read/write and random read/write results for RAIN-0 and RAIN-10 setups.

sizes. In the case of small requests, PVFS2 and GFS perform better due to their client-side cache which aids sequential prefetching. For sequential writes (Figure 7-b) *0FS* performs worse than PVFS2 and GFS in large block sizes, showing the overhead of the reliability protocols. In RAIN-10 and for average request sizes (512KB-1MB), *0FS* is similar to GFS. However, when the request size increases, the increased number of acknowledgments for the two-phase protocol incur a performance degradation of 15-20%. For small request sizes, performance is again lower due to the lack of client-side caching.

In the case of random I/O, Figures 7-c and 7-d, *0FS* performs better than both GFS and PVFS2, since the random workload minimizes the client-side caching effects. Random write for the RAIN-10 volume (Figure 7-d) shows practically no performance overhead compared to GFS, which has no reliability protocol. In the RAIN-0 case, the overhead of the protocol appears in the very large block sizes. Please note that, even in the case of RAIN-0, our prototype uses CIs and the two-phase protocol for writes since this is required to guarantee atomic updates of distributed data blocks.

Finally, in PostMark (Figure 8[1]) we observe that *0FS* mostly outperforms PVFS2 on RAIN-0. On the other hand, GFS maintains a lead in all cases, except in large requests, where *0FS* performs very close. We attribute this performance lead of GFS in metadata-intensive workloads to the use of the client cache for metadata and the fact that *0FS* needs to cross the kernel boundary 3-4 times per filesystem operation. As block size increases, this does not happen as often, and we are able to match GFS performance in large block sizes. The performance of GFS also shows that it performs asynchronous logging, flushing the metadata log to disk infrequently.

As mentioned, *0FS* does not use a client side cache. To examine the impact of the lack of a cache on *0FS* we show in Figure 9 the total data volume that reaches physical disks. First, we see that in *0FS* configurations this amount is higher by up to about 20%. Second, (not shown here) the amount of data for each request size remains

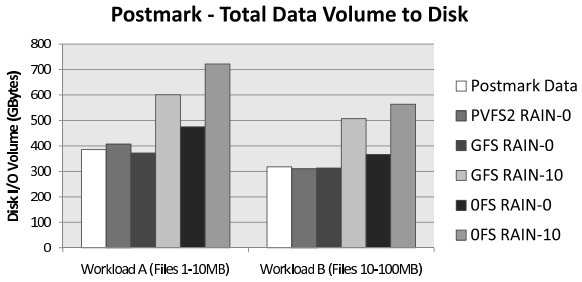


Figure 9: Postmark: Total data volume reaching disks.

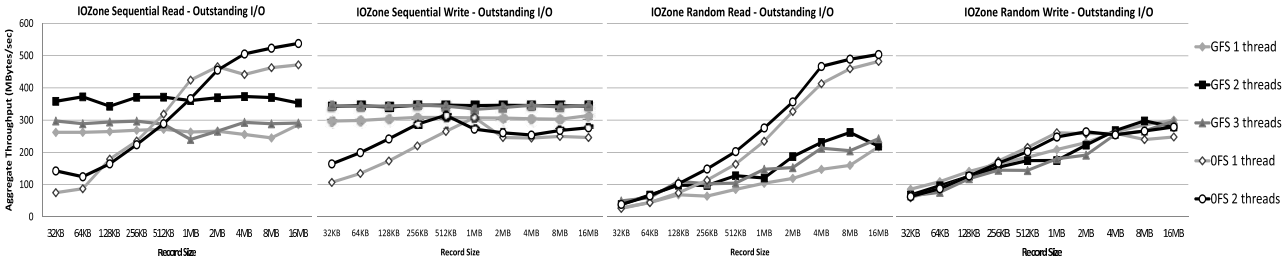
approximately the same. Given that GFS performs better only for smaller requests, we believe that the extra traffic is not a bottleneck that skews our results. On the other hand, the GFS cache affects request response time, especially for small requests, and since PostMark is latency-bound, we believe that this role of the client-cache results in the better GFS results for small request sizes. This effect is exacerbated by the use of TCP/IP as our communication subsystem, which not only incurs high latencies compared to state-of-the-art networks, but also exhibits problematic behavior, such as the Incast problem [49].

8.2 Overhead of availability

To examine the performance overhead of providing availability, we compare the performance of RAIN-0 vs. RAIN-10. This comparison is only applicable to GFS and *0FS*, since PVFS2 does not provide replication. For IOZone (Figure 7), we see that performance degrades by approximately a factor of two as expected, because of replication. In read requests however, both GFS and *0FS* perform better in RAIN-10 than RAIN-0 for large requests because of the efficient load-balancing of replica reads.

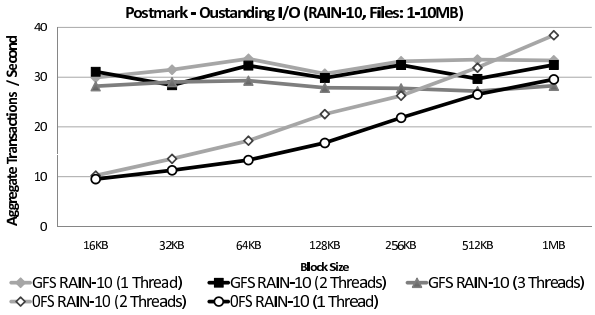
For postmark workloads (Figure 8) we observe also an average 20% performance degradation in the transaction rate, despite the fact that available disk throughput is effectively reduced to half. Overall we find that the performance impact of availability with *0FS* in metadata-intensive workloads is in the range of 15-20%.

[1]Note that transactions in this figure are PostMark-reported transactions and not related to *RIBD*'s operation.

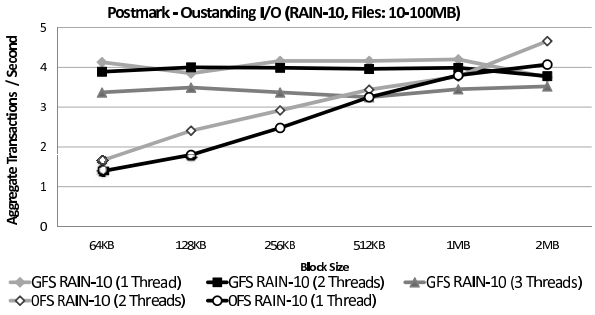


(a) Sequential Read (b) Sequential Write (c) Random Read (d) Random Write

Figure 10: IOzone sequential and random results (throughput vs. record size) for RAIN-10 with up to 3 threads per client.



(a) Workload A (RAIN-10 Setup)



(b) Workload B (RAIN-10 Setup)

Figure 11: PostMark aggregate transaction rate (trans/sec) for workloads A and B with up to 3 threads per client.

8.3 Impact of outstanding I/Os

To examine the impact of increasing the number of outstanding I/Os in the system we use IOZone with GFS and *0FS* on a RAIN-10 configuration and Postmark with workloads A and B on RAIN-0 and RAIN-10. In these experiments we have used multiple benchmark threads per client node, thus multiplying the load on the system and inducing more concurrent I/Os. IOZone supports multi-threaded mode, however Postmark does not have such an option so we executed multiple Postmarks in parallel on each node.

In Figure 10 we see that increasing the number of outstanding I/Os of IOZone from one to two, increases throughput for both GFS and *0FS* up to about 30%. We observe, however, that in the case of GFS, more than two outstanding I/Os per client can degrade performance (sequential reads) up to 20%.

FE	GFS		GFS		PVFS2		*0FS*		*0FS*	
RAIN	0		10		0		0		10	
Sys/Wait	S	W	S	W	S	W	S	W	S	W
[A] Server	11	32	14	39	10	19	12	0	15	0
[A] Client	17	36	20	35	5	0	10	37	12	33
[B] Server	15	25	17	32	17	16	15	0	17	0
[B] Client	23	34	25	32	8	0	11	32	13	27

Table 3: Average percentage of CPU time used by the system and for I/O wait in Postmark workloads A,B.

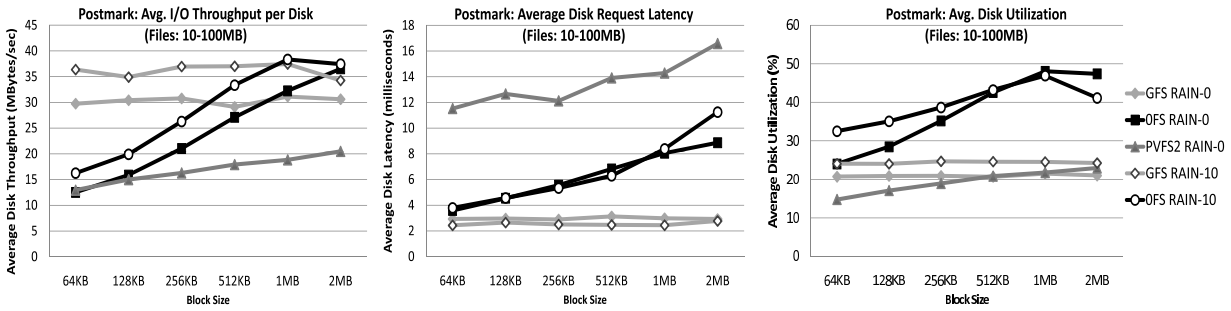
In the case of Postmark (Figure 11), we also observe that increasing the number of outstanding I/Os also increases throughput up to about 30% in GFS and *0FS*. On the other hand, PVFS2 does not seem to benefit from the increased concurrent I/Os as its performance remains the same as with the single thread. Finally, Figure 11 shows that GFS does not seem to scale with more than two threads per client. *0FS* results with more than two threads are not available because of stability issues in our prototype.

8.4 Impact on system resources

Now we examine system resource utilization in all systems we have measured.

CPU utilization: Table 3 shows the CPU utilization on the server and client sides for both PostMark workloads. We show only system and wait times, as user time is always less than 5% and in most cases less than 2%. First, we observe that overall system CPU utilization remains at low levels and up to 25% in the worst case, while I/O wait times can rise up to 40% both on the server and client side. However, the CPU has not been saturated in any of the experiments. Second, we note that all filesystems have similar system CPU utilizations on the disk server side. Finally, we note that on the client side, GFS incurs significantly higher CPU utilization, due to the use of a client-side cache.

Disk latency and utilization: Figure 12 shows the disk-level statistics during the PostMark runs, averaged



(a) Disk Throughput (MBytes/sec) (b) Disk Response Time (c) Disk Utilization

Figure 12: PostMark disk statistics for workload B. Values are averaged across all disks.

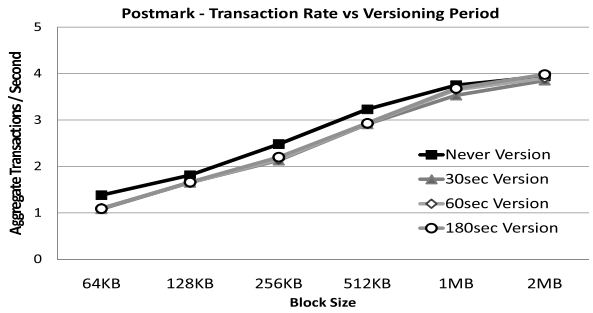


Figure 13: PostMark aggregate transaction rate for workload B and four versioning frequencies.

over the run time of the experiment and over all disks. Figure 12-a shows the average disk throughput for read and write requests. We observe that *0FS* reaches higher throughput. This is mainly due to the fact that *RIBD* is able to send to the disk larger requests than GFS. GFS requests go through the client side cache and are eventually issued at smaller sizes to the disk, resulting in lower throughput. Differences in Postmark disk request response time, shown in Figure 12-b, reflect longer seeks due to differences in block placement and access locality. *0FS* incurs higher latency than GFS because it does not use a client-side cache and it cannot fence metadata accesses. Thus, data and metadata requests alternate more frequently than in GFS, resulting in longer seek times. Finally, Figure 12-c shows average disk utilization. We see that *RIBD* results in up to double disk utilization compared to GFS and PVFS.

Finally, we conclude that the bottleneck in all file systems and workloads we measured is the networking layer, since neither the nodes' CPUs or the disks were saturated.

8.5 Versioning Protocol Overhead

To examine the overhead of the versioning protocol in *RIBD* we repeat the Postmark experiment with *0FS* using workload B on RAIN-10 (shown in Figure 8(b)) and four versioning frequencies: 30 seconds, 60 seconds, 180 seconds and never (i.e. no versioning). The results are shown in Figure 13. As shown, the maximum overhead of this protocol on the Postmark transaction rate is 21.7% for the 30 second version case with a block size of 64 KBytes. As the block size is increased to 1 MByte, the overhead of the protocol becomes less that 2%. We also find that lowering the versioning frequency improves performance slightly, since the versioning agreement protocol runs less frequently.

8.6 Summary

Overall, we find that although *RIBD*'s protocols for maintaining consistency affect system behavior and consume resources, performance and scalability, especially for larger requests, remains comparable to GFS and PVFS2. We also find that performance in small requests and metadata-intensive workloads is greatly enhanced by a client-side cache. Finally, we conclude that *RIBD*'s approach is better than existing solutions, since it offers stronger consistency guarantees with similar performance.

9 Conclusions

Cluster-based storage with commodity components is a promising alternative to scaling capacity and performance of future storage systems in a cost-effective manner. However, their decentralized nature poses important challenges, especially in terms of reliability and availability. Current solutions to this problem focus mostly at the file level and result in complex systems that are difficult to design, scale, and tune. In this paper we discuss how consistency issues can be addressed at the block level providing a simple abstraction. Our approach, *RIBD*, uses CIs, a lightweight transactional mechanism, agreement, versioning, and explicit locking, to address consistency of both replicas and metadata. We discuss in detail associated protocols and we implement a real prototype, showing that *RIBD* performs comparably to systems with weaker guarantees

References

[1] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Analysis and Evolution of Journaling File Systems," in *Proceedings of the USENIX Annual Technical Conference (USENIX '05)*, (Anaheim, CA), pp. 105–120, April 2005.

[2] J. Yang, P. Twohey, D. R. Engler, and M. Musuvathi, "Using Model Checking to Find Serious File System Errors," in *Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI 2004)*, (December 6-8, 2004, San Francisco, California, USA. USENIX Association), pp. 273–288, 2004.

[3] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "IRON File Systems," in *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, (Brighton, United Kingdom), pp. 206–220, October 2005.

[4] Compellent, "Storage Center Data Sheet." http://www.compellent.com/∼/media/ com/ Files/ Datasheets/DS_FT_021908.ashx.

[5] XIV Ltd., "Delivering the Thin Provisioning Advantage with XIV's Nextra Architecture White Paper." http://www.xivstorage.com/ materials/ white_papers/ nextra_thin_provisioning_white_paper.pdf.

[6] XIV Ltd., "Nextra Snapshot Implementation White Paper." http://www.xivstorage.com/ materials/ white_papers/ nextra_snapshot_white_paper.pdf.

[7] Enterprise Volume Management System, "evms.sourceforge.net."

[8] FreeBSD: GEOM Modular Disk I/O Request Transformation Framework, "http://kerneltrap.org/node/view/454."

[9] D. Teigland and H. Mauelshagen, "Volume managers in linux," in *Proceedings of USENIX 2001 Technical Conference*, June 2001.

[10] Sean Quinlan and Sean Dorward, "Venti: A New Approach to Archival Data Storage," in *Proceedings of FAST '02*, pp. 89–102, USENIX, Jan. 28–30 2002.

[11] R. English and S. Alexander, "Loge: A Self-Organizing Disk Controller," in *Proceedings of the Winter 1992 USENIX Conference*, (Berkeley, CA), The USENIX Association, 1992.

[12] R. Y. Wang, T. E. Anderson, and D. A. Patterson, "Virtual Log Based File Systems for a Programmable Disk," in *Proceedings of Operating Systems Design and Implementation (OSDI)*, pp. 29–43, 1999.

[13] J. Wilkes, R. A. Golding, C. Staelin, and T. Sullivan, "The HP AutoRAID Hierarchical Storage System," *ACM Transactions on Computer Systems*, vol. 14, pp. 108–136, Feb. 1996.

[14] M. Sivathanu, V. Prabhakaran, F. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Semantically-Smart Disk Systems," in *Proceedings of the FAST '03 Conference on File and Storage Technologies (FAST-03)*, USENIX Association, Apr. 2003.

[15] D. Stodolsky, M. Holland, I. William V. Courtright, and G. A. Gibson, "Parity-logging disk arrays," *ACM Trans. Comput. Syst.*, vol. 12, no. 3, pp. 206–235, 1994.

[16] T. Denehy, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Bridging the Information Gap in Storage Protocol Stacks," in *Proceedings of the USENIX Annual Technical Conference (USENIX '02)*, pp. 177–190, June 2002.

[17] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur, "PVFS: A Parallel File System For Linux Clusters," in *Proceedings of the 4th Annual Linux Showcase and Conference*, 2000.

[18] S. Soltis, G. Erickson, K. Preslan, M. O'Keefe, and T. Ruwart, "The Global File System: A File System for Shared Disk Storage," Oct. 1997.

[19] S. Chutani, O. T. Anderson, M. L. Kazar, B. W. Leverett, W. A. Mason, and R. N. Sidebotham, "The Episode file system," in *Proceedings of the USENIX Winter 1992 Technical Conference*, (San Fransisco, CA, USA), pp. 43–60, 1992.

[20] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck, "Scalability in the xfs file system," in *ATEC '96: Proceedings of the annual conference on USENIX Annual Technical Conference*, (Berkeley, CA, USA), pp. 1–1, USENIX Association, 1996.

[21] S. Tweedie, "Ext3, journaling filesystem," in *Presentation at Ottawa Linux Symposium*, (Ottawa Congress Centre, Canada), July 2000.

[22] E. K. Lee and C. A. Thekkath, "Petal: Distributed virtual disks," in *Proc. of The 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS7)*, (Cambridge, MA), pp. 84–92, Oct. 1996.

[23] C. Attanasio, M. Butrico, C. Polyzois, S. Smith, and J. Peterson, "Design and implementation of a recoverable virtual shared disk," Tech. Rep. IBM Research Report RC 19843, IBM T. J. Watson Research Center, Yorktown Heights, NY, 1994.

[24] D. J. Santry, M. J. Feeley, N. C. Hutchinson, and A. C. Veitch, "Elephant: The file system that never forgets," in *HOTOS '99: Proceedings of the The Seventh Workshop on Hot Topics in Operating Systems*, (Washington, DC, USA), p. 2, IEEE Computer Society, 1999.

[25] C. A. Soules, G. R. Goodson, J. D. Strunk, and G. R. Ganger, "Metadata Efficiency in Versioning File Systems," in *Proceedings of the FAST '03 Conference on File and Storage Technologies (FAST-03)*, (Berkeley, CA), The USENIX Association, Apr. 2003.

[26] A. Hisgen, A. Birrell, C. Jerian, T. Mann, and G. Swart, "New-value logging in the Echo replicated file system," Tech. Rep. 104, Xerox, Palo Alto CA (USA), 1993.

[27] K. A. Amiri et al., "Highly Concurrent Shared Storage," in *Proceedings of* 20^{th} *IEEE ICDCS Conference)*, (Taipe, Taiwan), IEEE Computer Society, Apr. 2000.

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

[29] J. Bonwick and B. Moore, "Zfs: The last word in file systems." http://opensolaris.org/os/community/zfs/ docs/zfs_last.pdf.

[30] W. D. Roome, "3dfs: A time-oriented file server," in *Proceedings of USENIX '92 Winter Technical Conference*, Jan. 1992.

[31] M. A. Olson, "The Design and Implementation of the Inversion File System," in *Proc. of USENIX '93 Technical Conference*, Jan. 1993.

[32] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom, "Plan 9 from Bell Labs," *Computing Systems, Summer, 1995.*, vol. 8, pp. 221–254, Summer 1995.

[33] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. N. Soules, and G. R. Ganger, "Self-Securing Storage: Protecting Data in Compromised Systems," in *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI-00)*, (Berkeley, CA), pp. 165–180, The USENIX Association, Oct. 23–25 2000.

[34] R. Hagmann, "Reimplementing the cedar file system using logging and group commit," in *SOSP '87: Proceedings of the eleventh ACM Symposium on Operating systems principles*, (New York, NY, USA), pp. 155–162, ACM, 1987.

[35] D. Hitz, J. Lau, and M. Malcolm, "File System Design for an NFS File Server Appliance," in *Proc. of the USENIX Winter 1994 Technical Conf.*, (San Fransisco, CA, USA), pp. 235–246, 17–21 1994.

[36] R. H. Patterson, S. Manley, M. Federwisch, D. Hitz, S. Kleiman, and S. Owara, "SnapMirror: File-System-Based Asynchronous Mirroring for Disaster Recovery," in *Proceedings of the FAST '02 Conference on File and Storage Technologies (FAST-02)*, (Berkeley, CA), pp. 117–130, USENIX Association, Jan. 28–30 2002.

[37] O. Krieger and M. Stumm, "Hfs: a performance-oriented flexible file system based on building-block compositions," *ACM Trans. Comput. Syst.*, vol. 15, no. 3, pp. 286–321, 1997.

[38] E. Zadok and J. Nieh, "FiST: A Language for Stackable File Systems," in *Proc. of the 2000 USENIX Annual Technical Conf.*, pp. 55–70, June 18–23 2000.

[39] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis, "Sinfonia: a new paradigm for building scalable distributed systems," in *SOSP '07: Proc. of 21st ACM SIGOPS Symposium on Operating systems principles*, (New York, NY, USA), pp. 159–174, ACM, 2007.

[40] J. Hartman, I. Murdock, and T. Spalink, "The Swarm Scalable Storage System," in *19th IEEE International Conference on Distributed Computing Systems (ICDCS '99)*, (Washington - Brussels - Tokyo), pp. 74–81, IEEE, May 1999.

[41] Y. Saito, S. Frolund, A. Veitch, A. Merchant, and S. Spence, "FAB: Enterprise storage systems on a shoestring," in *Proc. of the ASPLOS 2004*, Oct. 2004.

[42] F. Schmuck and R. Haskin, "GPFS: A Shared-disk File System for Large Computing Centers," in *USENIX Conference on File and Storage Technologies*, (Monterey, CA), pp. 231–244, Jan. 2002.

[43] D. Skeen, "Nonblocking commit protocols," in *SIGMOD '81: Proceedings of the 1981 ACM SIGMOD international conference on Management of data*, (New York, NY, USA), pp. 133–142, ACM, 1981.

[44] C. A. Thekkath, T. Mann, and E. K. Lee, "Frangipani: A Scalable Distributed File System," in *Proc. of the 16th Symposium on Operating Systems Principles (SOSP-97)*, pp. 224–237, Oct. 5–8 1997.

[45] K. W. Preslan, A. P. Barry, J. E. Brassow, G. M. Erickson, E. Nygaard, C. J. Sabol, S. R. Soltis, D. C. Teigl, and M. T. O'keefe, "A 64-bit, shared disk file system for Linux," in *16th IEEE Conference on Mass Storage Systems and Technologies (MSST '99)*, Mar. 1999.

[46] PVFS2 Project, "PVFS2 Home Page." http://www.pvfs.org.

[47] W. D. Norcott and D. Capps, "IOzone Filesystem Benchmark." http://www.iozone.org.

[48] J. Katcher, "PostMark: A New File System Benchmark." http://www.netapp.com/tech_library/3022.html.

[49] A. Phanishayee, E. Krevat, V. Vasudevan, D. G. Andersen, G. R. Ganger, G. A. Gibson, and S. Seshan, "Measurement and Analysis of TCP Throughput Collapse in Cluster-based Storage Systems," in *Proc. of the 6th USENIX Conference on File and Storage Technologies (FAST08)*, pp. 175–188, 2008.