Evaluating the Price of Consistency in Distributed File Storage Services

José Valerio, Pierre Sutra, Étienne Rivière, Pascal Felber

Evaluating the Price of Consistency in Distributed File Storage Services

José Valerio, Pierre Sutra, Étienne Rivière, and Pascal Felber

University of Neuchâtel,
Switzerland

Abstract. Distributed file storage services (DFSS) such as Dropbox, iCloud, SkyDrive, or Google Drive, offer a filesystem interface to a distributed data store. DFSS usually differ in the consistency level they provide for concurrent accesses: a client might access a cached version of a file, see the immediate results of all prior operations, or temporarily observe an inconsistent state. The selection of a consistency level has a strong impact on performance. It is the result of an inherent tradeoff between three properties: consistency, availability, and partition-tolerance. Isolating and identifying the exact impact on performance is a difficult task, because DFSS are complex designs with multiple components and dependencies. Furthermore, each system has a different range of features, its own design and implementation, and various optimizations that do not allow for a fair comparison. In this paper, we make a step towards a principled comparison of DFSS components, focusing on the evaluation of consistency mechanisms. We propose a novel modular DFSS testbed named FlexiFS, which implements a range of state-of-the-art techniques for the distribution, replication, routing, and indexing of data. Using FlexiFS, we survey six consistency levels: linearizability, sequential consistency, and eventual consistency, each operating with and without close-to-open semantics. Our evaluation shows that: *(i)* as expected, POSIX semantics (i.e., linearizability without close-to-open semantics) harm performance; and *(ii)* when close-to-open semantics is in use, linearizability delivers performance similar to sequential or eventual consistency.

1 Introduction

Distributed file storage services (DFSS) offer a unified filesystem view of unstructured distributed data stores. As for any distributed storage service, the expected properties of a DFSS are consistency, availability, and tolerance to partitions. The CAP theorem [1] states that a distributed storage system can fully support at most two of these three properties simultaneously. Partition tolerance is usually considered essential for a DFSS, as data centers may be temporarily disconnected in a large-scale distributed setting and such events must be supported. As a result, developers of DFSS usually decide on a tradeoffs between availability and consistency.

Historically, DFSS designs have focused on providing the POSIX strong model of consistency [2]. The need for planetary-scale and always available services, and

thus the shift to geographically distributed platforms and cloud architectures, has led DFSS designs to focus more heavily on availability and weaker consistency levels. By introducing caching mechanisms and the *close-to-open* semantics, the Andrew filesystem (AFS) [3] was one of the first systems to offer a consistency level weaker than POSIX. Most operations in systems such as HDFS [4] or GoogleFS [5] are sequentially consistent. However, both systems are built around a central metadata server. Schvachko [6] recently pointed out that this approach is inherently not scalable because the metadata server cannot handle massive parallel writes and the physical limits of a central metadata server design hits the petabyte barrier, i.e., the system cannot address more than 10^{15} bytes. On the other hand, flat storage systems like Cassandra or Dynamo [7, 8] propose an even weaker consistency level: eventual consistency. This relieves designers from the need for a central metadata server. Some systems [9–11] implement a filesystem interface on top of an eventually consistent storage system. However, to the best of our knowledge, none has gained wide acceptance.

Enabling further research on DFSS to scale and break the petabyte barrier requires developers to understand and be able to compare systematically the multiple components of a design. These components include data distribution and replication (and associated consistency guarantees), request routing, data indexing and querying, or access control. Performing a fair comparison of these aspects as supported by existing DFSS implementations is difficult because of their inherent differences. Indeed, these systems propose not only different filesystem consistency levels (FSCLs), but they also feature different base performance and optimization levels, which largely depend on the programming language, environment, runtime, etc.

In this paper, we make a step toward allowing the systematic comparison of DFSS designs. We instantiate our approach by isolating and evaluating the impact of the FSCL on performance. We make the following contributions:

- We depict the construction of a filesystem on top of a regular key/value store with the simple addition of the *compare-and-swap* primitive.
- We present a clear typology of the different FSCLs and categorize existing implementations accordingly.
- We compare empirically FSCLs by instantiating them into a novel DFSS testbed named FlexiFS. Our testbed is modular and implements a range of state-of-the-art techniques for the distribution, replication, routing, and indexing of data.

Our main findings are the following: *(i)* as expected, POSIX semantics (i.e., atomicity without close-to-open semantics) harms performance; and *(ii)* when close-to-open semantics is in use, atomicity delivers performance similar to sequential or eventual consistency.

The remainder of this paper is organized as follows: We describe the design of FlexiFS in Section 2. Section 3 introduces FSCLs and their corresponding implementations in FlexiFS. We present several benchmark results that evaluate each level in Section 4. Section 5 surveys related work. We close in Section 6.

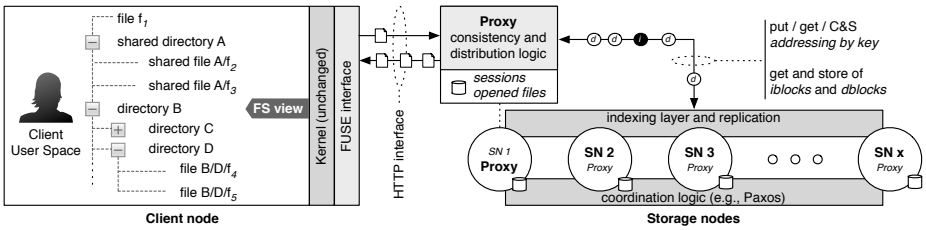


Fig. 1. General Architecture of FlexiFS.

2 Testbed Design

FlexiFS is a distributed file storage service (DFSS) offering a transparent filesystem interface. Figure 1 illustrates its general architecture. FlexiFS has been built in a modular way to allow evaluation of different choices of DFSS designs. A typical deployment contains two sets of nodes: *storage nodes* implement a distributed flat storage layer, while *client nodes* present a filesystem abstraction to the users, and store files and folders hierarchies on the storage nodes.

A client node accesses the filesystem through a *filesystem in user space* (FUSE) implementation [12]. FUSE is a loadable kernel module that provides a filesystem API to the user space. It lets non-privileged users create a filesystem without writing any kernel code. In FlexiFS, each access to the filesystem is transformed into a Web service request and routed toward a *proxy* node that acts as an entry point to the distributed storage. The proxy redirects requests to the adequate storage node(s), which store or return data blocks.

2.1 Proxy

The role of the proxy is to hide both the topology of the distributed storage and the operation logic from the client. In FlexiFS, any storage node can act as a proxy. When a client executes an operation and contacts a proxy via its Web service interface, the proxy accesses the underlying storage system, executes the operation, and returns the result to the client.

The storage layer is essentially a key/value store extended with a *compare-and-swap* primitive. Devising a filesystem on top of this interface is a contribution of our work. The operations of the interface are as follows:

- $put(k, v)$: writes the value v for key k,
- $get(k)$: returns the data stored for key k,
- $C\&S(k, u, v)$: executes a compare-and-swap on key k with old value u and new value v.[1]

Depending on the FSCL in use in FlexiFS, the semantics of the above interface may change. For instance, $C\&S()$ is not atomic under eventual consistency. We detail how FlexiFS implements this interface in Section 3.

[1] $C\&S(k, u, v)$ checks whether the stored value is still u, and if so, replaces u by v; in any case the old value is returned.

2.2 Distributed Storage Layer

FlexiFS is modular and decouples the filesystem logic from the actual storage. The storage layer supports data indexing and provides the API described in the previous section. Due to its modular design, FlexiFS is able to use different indexing and storage layers, such as a multi-hop DHT or a central server. Below, we detail the design common in flat storage layers [7, 8] that we use in this paper.

FlexiFS's indexing and storage layer is a simple yet efficient one-hop DHT structured as a ring that relies on consistent hashing to store and locate data. Figure 1 presents its general architecture. It supports the following features:

Routing. For performance reasons and in order to reduce noise in our experiments, we have chosen a one-hop routing design, i.e., every node knows all other nodes in the ring.

Elasticity. Upon joining, a node chooses a random identifier along the ring and fetches the ring structure from some other DHT node. It then informs its two direct neighbors that it is joining.

Storage. FlexiFS uses consistent hashing to assign blocks to nodes [13] with replication factor r: a block with a key k is stored at the r nodes whose identifiers follows k on the ring.

Failure detection. Each node periodically checks the availability of its closest successor on the ring, and repair mechanisms are triggered upon a lack of response within a timeout.

A gossiping mechanism spreads topological changes throughout the ring. Each node notifies its closest neighbor whenever it learns about a leave/join event. If the time to spread a message along the ring is shorter than the time between two leave/join events, this mechanism is guaranteed to maintain the ring topology. In our experience, such an assumption is reasonable for a deployment size of less than a few hundred storage nodes (our typical testbed size).[2]

2.3 Filesystem

Like most contemporary DFSS, FlexiFS decouples metadata from data storage. For each file, an inode block (`iblock` hereafter) contains the metadata information about the file, e.g., size and user/group ownership. One or more data blocks (`dblock`) hold the content of the file.

FlexiFS provides several hooks to tune how files are stored. Figure 2 illustrates our current design: `dblocks` are of constant and configurable size. The current size of `dblocks` is 128 kB, corresponding to the default maximal block size for the FUSE interface. `iblocks` simply list the `dblocks` of the corresponding files. Compared to the typical redirection-based architecture of Unix, the two above mechanisms help in reducing the network overhead [5].

Both `iblocks` and `dblocks` are represented as elements of the same key/value store, where they get replicated according to the different consistency models.

[2] The probability that faults partition the ring is small. Indeed, we note that at the considered scale, the mean time between failures divided by the number of nodes is orders of magnitude smaller than the time to spread a message throughout the ring.

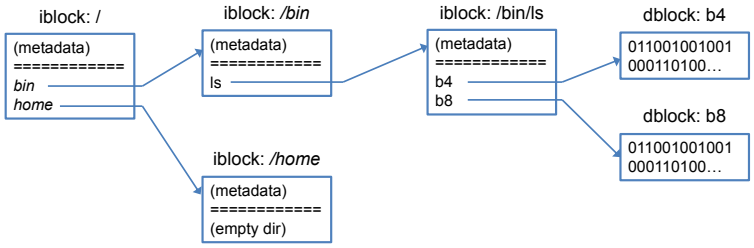


Fig. 2. Example of a filesystem structure stored in FlexiFS.

```
int(* create )(const char *, mode_t, struct fuse_file_info *);
int(* open )(const char *, struct feuse_file_info *);
int(* read )(const char *, char *, size_t, off_t, struct fuse_file_info *);
int(* write )(const char *, const char *, size_t, off_t, struct fuse_file_info *);
int(* close)(const char *);
int(* rename )(const char *, const char *);
int(* statfs )(const char *, struct statvfs *);
```

Fig. 3. Excerpt of the FUSE interface implemented by FlexiFS.

Only `iblocks` are mutable. The key of a `dblock` is equal to the hash of its content. This ensures good balancing of the data across storage nodes in order to deliver aggregate performance and increased fault tolerance. In case of an `iblock`, the proxy generates a unique key at creation time.

2.4 File Operations

FlexiFS implements the complete FUSE interface. We present an excerpt in Figure 3 and detail below the most important operations.[3]

Create. Upon the creation of a file (or directory), the proxy first stores a corresponding `iblock` in the distributed storage. Then, it executes $C\&S()$ on the parent directory to add the file. Performing a $C\&S()$ operation ensures that no two clients create the same file concurrently. If the file was concurrently created, the proxy returns an error to the client.

Open. To open an existing file (or directory), the proxy follows the graph structure of the filesystem and invokes the *get*() operation to retrieve the corresponding `iblock` from the storage interface. Once the `iblock` is fetched, the proxy checks that permissions are correct and returns an appropriate value to the client.

Read. The proxy first retrieves the `iblock` from the storage system. Once the `iblock` is known, the proxy also knows all the `dblocks` attached to it. Hence, to retrieve the content of the file, the proxy fetches the `dblocks` from the storage system by invoking the *get*() operation in parallel.

[3] Because an open file may acquire multiple reference (e.g., after a *fork*()), there is no *close*() operation in the FUSE interface, but instead *flush*() and *release*(). The former is called every time a descriptor referencing the file is closed; the latter once all file descriptors are closed and all memory mappings are unmapped. To simplify exposition, we shall consider in this paper that *close*() is part of the FUSE interface.

Write. The proxy first retrieves the `iblock` of the file and produces the new `dblocks`. It uses the storage layer's *put*() operation to insert (in parallel) the new `dblocks` in the distributed storage. Notice that because `dblocks` are content-addressed and immutable, every modification that produces a `dblock` leads to the creation of a new `dblock` with a different key. Then, the proxy uses *C&S*() to update the `iblock` corresponding to a file. If the `iblock` changed meanwhile, the proxy has to recompute (if necessary) the `dblocks`, as well as an updated version of the `iblock`; then it re-executes *C&S*(). This last sequence of operations is executed until the *C&S*() succeeds. Since a *write*() operation may access any offset of the file, the above mechanism is necessary to avoid a lost update phenomena when two clients concurrently write the file.

Close. Upon the closing of a file, the proxy checks that the file still exists. If the file does not exist, the proxy returns an error to the client.

Rename. If the source and target parent directories are the same, the proxy attempts updating the `iblock` of the parent directory. In case they are different, the proxy first tries adding the file to the target directory, then it attempts removing the file from the source directory. All attempts are perform with *C&S*(). If *C&S*() fails at some point, the proxy returns an error to the client.[4]

3 Consistency

An important design aspect for a DFSS is defining the semantics of sharing, i.e., how clients accessing simultaneously the same file observe modifications by other clients. FlexiFS has several built-in sharing semantics and corresponding implementations, which we describe in the remainder of this section. Additional semantics can be easily added thanks to FlexiFS's modular design.

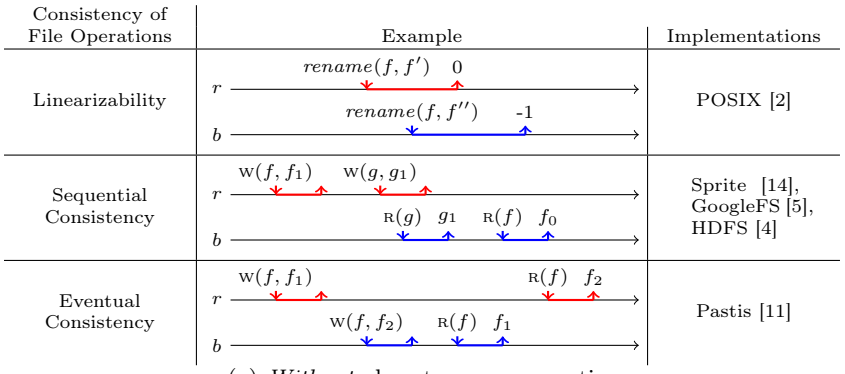
3.1 Overview

FlexiFS classifies the semantics of sharing with (i) the consistency level that governs the FUSE interface; and (ii) the use (or not) of the close-to-open semantics. The combination of these two parameters defines a filesystem consistency level (FSCL). In what follows, we list the various consistency levels FlexiFS supports at the FUSE interface and their respective implementation. Further, we introduce the close-to-open semantics.

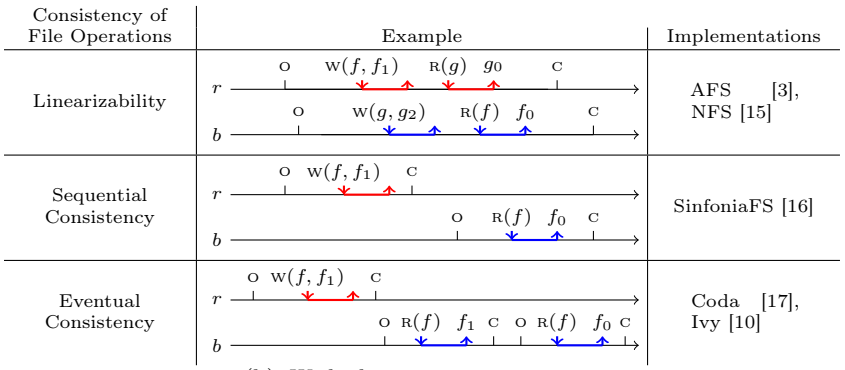
3.2 Consistency of the FUSE Interface

The three operations available at the storage layer implement accesses to the filesystem. As a consequence, the consistency of the distributed storage governs consistency of the FUSE interface. The FUSE interface is linearizable (resp. sequentially, eventually consistent) when operations at the storage level are linearizable (resp. sequentially, eventually consistent).

[4] Even if *C&S*() is atomic, the renamed file might end up in both source and target directories. Renaming is strictly atomic in POSIX semantics. We note however that such a behavior is admissible in certain systems (e.g., Win32).



(a) *Without* close-to-open semantics



(b) *With* close-to-open semantics

Fig. 4. Filesystem Consistency Levels. *(operation* $\text{W}(f, v)$ *means a write to file* f *with value* v*;* $\text{R}(f)$ *is a read on* f*;* O *and* C *respectively opens and closes all files accessed during the session.)*

Linearizability. The most powerful synchronization level for processes in a distributed environment is obtained through the use of atomic, or *linearizable*, objects [18]. A linearizable object is a shared object that provides the illusion of being accessed locally. More precisely, this consistency level states that each operation takes effect instantaneously at some point in real time, between its invocation and response.

Figure 4(a) presents an execution of linearizable operations. The blue (b) client renames file f to f'. Concurrently, the red (r) client renames file f to f''. Since operations are linearizable, one of the two accesses must fail.

To implement linearizability in FlexiFS, we use Paxos [19], which provides a consensus primitive for unreliable nodes. On top of consensus, we implement a replicated state machine executing the three operations listed in Section 2.1. Notice that, because `dblocks` are immutable, they are trivially linearizable. Hence, to improve performance, we execute a simpler algorithm in that case: operations *put*() and *get*() access a majority of replicas, respectively storing and fetching the content from it.

Sequential Consistency. Under sequential consistency, "the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program" [20]. Sequential consistency is weaker than linearizability. In particular, this consistency level is not composable [18]: even if each file is sequentially consistent, the filesystem as a whole is not sequentially consistent (this is also called the *hidden channel* problem). We illustrate this issue in Figure 4(b) (middle). In this figure, accesses with respect to file f are sequentially consistency, and similarly this property holds for g. However, the execution (i.e., when we consider both f and g as a whole) is not sequentially consistent.

FlexiFS implements sequential consistency using primary replication. For each `iblock`, a primary replica is elected. Upon a $put(k, v)$ call, the primary for key k sends to all replicas the value v and then waits until a majority of replicas acknowledges the reception before returning to the proxy. To execute a $get(k)$ call, the proxy accesses any replica of k that contains the version it previously read, or a newer version (this applies only to `iblocks`). To execute $C\&S(k, u, v)$, the primary tests locally if the old value equals u. If it is the case, it executes a $put(k, v)$ and returns the old value to the proxy. A perfect failure detector [21] ensures the safety and liveness of the above mechanisms.

Eventual Consistency. Under eventual consistency [22], there must exist a monotonically growing prefix of updates on which correct replicas agree. Since there is no assumption on the time of convergence, eventual consistency does not offer any guarantee on the return value of non-stable operations (that do not belong to the common prefix).

Figure 4(b) (bottom) depicts a run under eventual consistency. In this figure, both b and r clients write then read file f. Because the r client reads version f_2 while the b client reads version f_1, no linearization of the four operations can satisfy the returned values.

We implement eventual consistency in FlexiFS using version vectors and the "last writer wins" approach [23]. This optimistic replication schema works as follows: Each version of the `iblock` is timestamped with a version vector. Upon updating the value of an `iblock` (via $put()$ or $C\&S()$), the proxy contacts one of the `iblock` replicas. This replica atomically increments its local vector clock, timestamps the `iblock` with it, and returns to the proxy. Replicas then converge using an anti-entropy protocol. If two versions of some `iblock` are concurrent, we apply the "last writer wins" approach. Concurrent operations are totally ordered according to the identifier of the replicas that emitted them. Upon a read access, a proxy simply returns a version stored at some replica.

3.3 Close-to-Open Semantics

Under POSIX semantics, almost all file operations shall be linearizable [2, page 58]. In particular, a read shall see the effects of all previous writes performed on

the same file. The CAP impossibility result [1] tells us that such a constraint hinders the scalability of a DFSS.

By introducing the notion of *file session*, close-to-open semantics [24] aim at reducing the amount of synchrony required to access a shared file. A file session is a sequence of read and/or write operations enclosed between an *open* and a *close* invocation. It has the following properties [25]:

(1) Writes to an open file are visible to the client but are invisible to remote clients having the same file opened concurrently.
(2) Once the file is closed, changes are immediately visible to sessions that are starting afterwards.

Since operations execute in isolation and either all writes or none execute, these sharing semantics are close to the familiar notion of transaction. Notice, however, that close-to-open semantics apply the last writer wins rule: two concurrent updates do not abort, one of them is simply overwritten.

The above definition of close-to-open consistency has been formulated with atomicity in mind. One can actually combine close-to-open semantics with sequential consistency and eventual consistency as well. For sequential consistency, rule (2) is replaced by:

(2a) There exists a sequential ordering of the sessions such that (i) for every read in a session, there exists a matching write prior to it, and (ii) reads are causally ordered on the same client.

For eventual consistency, rule (2) is replaced by:

(2b) If at some point in time no more changes occurs, then eventually all sessions observe the same state of the file.

Figure 4(b) illustrates the combination of close-to-open semantics with linearizability, sequential, and eventual consistency. Obviously, if a single read or write operation is executed per session, the consistency level per operation defines how sessions behave. In other words, close-to-open semantics have no effect (e.g., middle row in Figure 4(b)). Now, when a client executes multiple operations or opens multiple files at the same time, the filesystem is neither linearizable nor sequentially consistent. For instance, execution depicted at the top row in Figure 4(b) is admissible. Under close-to-open semantics, linearizability is stricter than sequential consistency, which is itself stricter than eventual consistency (last two rows in Figure 4(b)).

When FlexiFS implements the close-to-open semantics, the proxy is stateful and keeps track of the files opened by each of its clients. Therefore, a client has to access the same proxy during a file session. In more details, upon a successful call to $open(f)$ the proxy records the `iblock` of f. This `iblock` is used for all the operations during a file session: a $read()$ operation accesses the `dblocks` indexed by the `iblock`, and a write operation changes only the cached version. When the client closes file f, the proxy stores the `iblock` of f using $put()$. It can then forget that f was opened by the client.

3.4 Consistency of the Filesystem

A filesystem consistency level (FSCL) is obtained by the combination of a consistency level governing file operations and the use or not of the close-to-open semantics. This leads to six *different* FSCLs. In Figure 4, we list in the last column one or more matching implementations for each level.

POSIX semantics is obtained when file operations are atomic and close-to-open is not supported. To implement sequential consistency, Sprite [14] relies on a cache consistency manager while GoogleFS [5] and HDFS [4] make use of a leasing mechanism. NFS [15] implements the consistency level offered in Andrew Filesystem [3]: the close-to-open semantics is respected and metadata operations are atomic. Sinfonia [16] supports mini-transactions, a generalized form of compare-and-swap operation. To advocate for this paradigm, the authors of Sinfonia built a filesystem: SinfoniaFS. This filesystem implements sequential consistency with close-to-open semantics. Ivy [10] and Pastis [11] implement an eventually consistent DFSS, respectively, with and without close-to-open semantics.

4 Evaluation

In this section, we present experimental results obtained using FlexiFS, where we observe empirically and in isolation the tradeoffs between sharing semantics and performance in DFSS designs.

4.1 Experimental Settings

All tests were performed on a cluster of 8-core virtualized Xeon 2.5 Ghz servers running Ubuntu 12.04 GNU/Linux and connected by a 1 Gbps switched network. We use 3 to 7 servers for the storage layer and one client. Our implementation uses the Lua programming language (http://www.lua.org/) and leverages the Splay framework and libraries [26]. Bindings to the FUSE C APIs employ the `luafuse` library (http://code.google.com/p/luafuse/). The FlexiFS implementation is modular and easy to modify. The conciseness of Lua and the use of Splay allow the whole implementation to be less then 2,000 lines of code (LOC). In particular, the code to support each FSCL is very concise and easy to extend, e.g., 62 LOC for sequential consistency, and 160 LOC for eventual consistency.

In what follows, we explore the impact of each FSCL on the cost of `iblock` operations, and file operations. We also investigate the impact of the replication factor on performance. All experimental results are averaged over 10^3 operations, and we present standard deviations when appropriate.

4.2 Benchmarks

Metadata Operations In Figure 5(a), we experiment the insertion of a novel `iblock` in the storage system when both the FSCL and the size of the inserted block vary. For the sake of comparison, results are normalized by the time required for executing a *noop*() RPC operation carrying a payload that equals the size of the block.

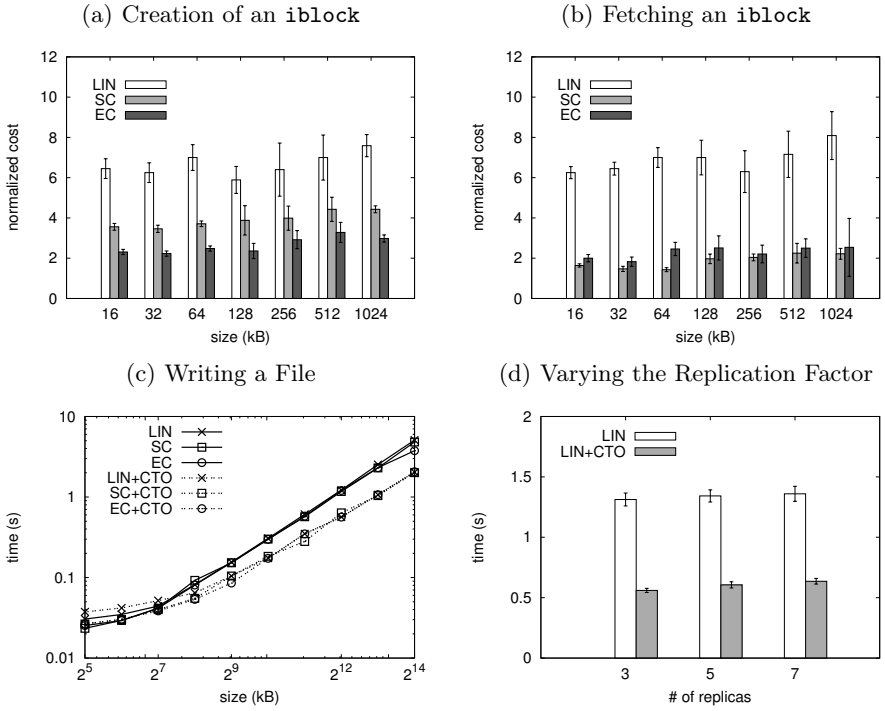


Fig. 5. Evaluating the Price of Consistency. *(LIN, SC and EC stand for Linearizable, Strong Consistency and Eventual Consistency FSCL models, respectively. CTO stands for close-to-open semantics.)*

We observe in Figure 5(a) that eventual consistency is the cheapest FSCL as it costs around 2 times more than the baseline RPC call. This is expected since a call to $C\&S()$ under eventual consistency requires 2 roundtrips: one to go from the client to the proxy, then one to go from the proxy to a replica. Sequential consistency costs 4 roundtrips: once the primary is reached, the update must reach a quorum of replicas. For linearizability, 2 more roundtrips for the "propose" phase of the Paxos algorithm are executed, leading to 6 times the baseline cost.

Our second experiment evaluates the cost of fetching an `iblock` from the file storage. We report the results in Figure 5(b). Under linearizability, a *get*() operation has an identical cost to a $C\&S()$ operation, since both operations go through the replicated state machine. On the other hand, the cost of sequential consistency is reduced because the proxy can access any replica to fetch the `iblock` content. Therefore, performance is in that case identical to eventual consistency.

File Operations Figure 5(c) depicts the time required to write a complete file at the client side using the FUSE interface. Both scales in this figure are logarithmic. Eventual consistency is the fastest FSCL when either *(i)* close-to-open semantics

is not used, or *(ii)* there is a single `dblock` to write, i.e., less than 128 kB in our settings. The POSIX semantics of sharing (i.e., linearizability without close-to-open semantics) performs the worst in this respect. When the size of the file reaches 128 kB, the use of close-to-open semantics leads to better performance (between 2 and 3 times faster). Below 128 kB, close-to-open semantics pays the cost of the necessary *open*() and *close*() operations. All FSCLs offering close-to-open semantics reach similar performance when more than 4 MB of data are written. Read operations over a file (not reported here) follow a similar pattern.

Impact of the Replication Factor Our last experiment measures the impact of the replication factor on performance. In this experiment, a client writes a file of 4 MB under linearizability. We vary both the replication factor of FlexiFS, and the use or not of the close-to-open semantics. Figure 5(d) depicts our results. In this figure, we observe that increasing the replication factor has a small impact on performance: below 3% without close-to-open semantics, and 7% with. Paxos is the most demanding consistency control algorithm we have implemented. Thus, this result shows that the filesystem consistency level is contributing more than the replication factor to DFSS performance.

5 Related Work

Several papers discuss the performance, consistency, and semantics tradeoffs in DFSS designs. The Andrew file system (AFS) [3] introduced caching mechanisms and the close-to-open semantics for both files and directories. This was inspired by earlier designs such as Locus [27], which relied on a strict—but costly and inefficient—POSIX semantics. Since its second version, the Network File System (NFS) [15] also implements the close-to-open semantics; its fourth version distinguishes data from metadata management, a separation that has been adopted by all DFSS designs since then.

OceanStore [28] is a flat data storage system that provides both eventually consistent and atomic operations. OceanStore follows a design close to the eventually serializable data storage [22]: an application may emit two types of file operations, *weak* and *strong*. A weak operation is tentative and executes on any replica; a strong operation waits until replicas agree on some total ordering of the operations. GoogleFS [5] uses a central server for storing metadata. CFS [9] builds a single-user file system by storing content-addressable blocks in the Chord DHT [29]. Ivy [10] extends this design by allowing a predefined group of users to access a shared file system. Content blocks are stored in the Chord DHT and each writer maintains its own modification log, implementing read-your-write semantics and eventual consistency. In [30], the authors propose to build centralized metadata storage services for DFSS, providing linearizability guarantees using the Paxos [19] consensus algorithm while maintaining high availability.

The authors of Pastis [11] compare close-to-open against read-your-write semantics. Levy [25] surveys DFS designs and four different types of file sharing semantics: POSIX, close-to-open, immutable files, fully transactional semantics,

and survey corresponding implementations. The use of a modular framework for evaluating design choices and establishing performance/tradeoffs in systems software design has been successfully used in various domains. Examples include virtual machines construction [31] or CORBA-based Middleware [32].

6 Conclusion

This paper depicts a study of the impact of a filesystem consistency level (FSCL) on the performance of a distributed filesystem service (DFSS). While the FSCL offered by a DFSS has fundamental impact on performance, it is difficult to systematically evaluate this impact in isolation from other design aspects, due to the design and implementation diversity of existing systems. This paper presents FlexiFS, a framework for the systematic evaluation of DFSS aspects. In more details, we depict a filesystem interface to users and leverage a set of servers implementing a fully distributed storage layer for both data and metadata. We implement three forms of consistency: linearizability, sequential consistency and eventual consistency, together with and without close-to-open semantics. Remarkably, a DFSS providing all these FSCLs can be supported with the simple addition of a *compare-and-swap* primitive to a regular key/value store. Our experimental results establish that linearizability under the close-to-open semantics is a sound design choice and a good compromise between operational semantics and performance, while illustrating the tradeoffs offered by the other design options. FlexiFS has a modular design and we plan on investigating further aspects of DFSS pertaining to indexing, client interaction, and semantics. We also plan on releasing FlexiFS as a part of the open-source Splay framework [26].

Acknowledgements

This work is sponsored in part by European Commission's Seventh Framework Program (FP7) under grant agreement No. 318809 (LEADS), and the Swiss National Foundation under agreements No. 200021-127271/1 and CRSII2-136318/1.

References

1. Gilbert, S., Lynch, N.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News **33**(2) (2002) 51–59
2. IEEE, The Open Group: Standard for Information Technology-Portable Operating System Interface (POSIX) System Interfaces (2004)
3. Howard, J.H., Kazar, M.L., Menees, S.G., Nichols, D.A., Satyanarayanan, M., Sidebotham, R.N., West, M.J.: Scale and performance in a distributed file system. ACM Trans. Comput. Syst. **6**(1) (February 1988)
4. The Apache Software Foundation: The Hadoop Distributed File System (2012)
5. Ghemawat, S., Gobioff, H., Leung, S.T.: The google file system. In: SOSP. (2003)
6. Shvachko, K.V.: HDFS Scalability: The Limits to Growth. USENIX login **35**(2) (April 2010)
7. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. SIGOPS Oper. Syst. Rev. **44**(2) (April 2010)

8. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: amazon's highly available key-value store. In: SOSP. (2007)
9. Dabek, F., Kaashoek, M.F., Karger, D., Morris, R., Stoica, I.: Wide-area cooperative storage with CFS. In: SOSP. (2001)
10. Muthitacharoen, A., Morris, R., Gil, T.M., Chen, B.: Ivy: a read/write peer-to-peer file system. In: OSDI. (2002)
11. Busca, J.M., Picconi, F., Sens, P.: Pastis: a highly-scalable multi-user peer-to-peer file system. In: Euro-Par. (2005)
12. File System in User Space: http://fuse.sourceforge.net/
13. Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., Lewin, D.: Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In: STOC. (1997)
14. Nelson, M.N., Welch, B.B., Ousterhout, J.K.: Caching in the sprite network file system. ACM Trans. Comput. Syst. **6**(1) (February 1988)
15. Sun Microsystems, Inc.: NFS: Network file system protocol specification. RFC 1094, Network Information Center, SRI International (March 1989)
16. Aguilera, M.K., Merchant, A., Shah, M., Veitch, A., Karamanolis, C.: Sinfonia: a new paradigm for building scalable distributed systems. In: SOSP. (2007)
17. Satyanarayanan, M., Kistler, J.J., Kumar, P., Okasaki, M.E., Siegel, E.H., Steere, D.C.: Coda: A highly available file system for a distributed workstation environment. IEEE Trans. Comput. **39**(4) (April 1990)
18. Herlihy, M., Wing, J.: Linearizability: a correcteness condition for concurrent objects. ACM Trans. Prog. Lang. and Sys. **12**(3) (July 1990)
19. Lamport, L.: The part-time parliament. ACM Trans. Comp. Sys. **16**(2) (1998)
20. Lamport, L.: How to make a correct multiprocess program execute correctly on a multiprocessor. IEEE Trans. Comput. **46**(7) (1997) 779–782
21. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal of the ACM **43**(2) (1996) 225–267
22. Fekete, A., Gupta, D., Luchangco, V., Lynch, N., Shvartsman, A.: Eventually-serializable data services. Theoretical Computer Science **220** (1999)
23. Saito, Y., Shapiro, M.: Optimistic replication. Computing Surveys **37**(1) (2005)
24. Saltzer, J.H., Kaashoek, M.F.: Principles of Computer System Design: An Introduction. Morgan Kaufmann Publishers Inc. (2009)
25. Levy, E., Silberschatz, A.: Distributed file systems: concepts and examples. ACM Comput. Surv. **22**(4) (December 1990)
26. Leonini, L., Rivière, E., Felber, P.: SPLAY: Distributed systems evaluation made simple. In: NSDI. (2009)
27. Walker, B., Popek, G., English, R., Kline, C., Thiel, G.: The LOCUS distributed operating system. In: SOSP. (1983)
28. Kubiatowicz, J., Bindel, D., Chen, Y., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Weimer, W., Wells, C., Zhao, B.: Oceanstore: An architecture for global-scale persistent storage. In: ASPLOS. (2000)
29. Stoica, I., Morris, R., Liben-Nowell, D., Karger, D.R., Kaashoek, M.F., Dabek, F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup protocol for internet applications. IEEE/ACM Trans. Net (1) (feb 2003)
30. Stamatakis, D., Tsikoudis, N., Smyrnaki, O., Magoutis, K.: Scalability of replicated metadata services in distributed file systems. In: DAIS. (2012)
31. Geoffray, N., Thomas, G., Lawall, J., Muller, G., Folliot, B.: VMKit: a substrate for managed runtime environments. In: VEE. (2010)
32. PolyORB Middleware Technology: http://libre.adacore.com/tools/polyorb