Loosely Time-Synchronized Snapshots in Object-Based File Systems

Jan Stender, Mikael Högqvist, Björn Kolbeck
Zuse Institute Berlin, Germany
E-mail: {stender, hoegqvist, kolbeck}@zib.de

Abstract

A file system snapshot is a stable image of all files and directories in a well-defined state. Local file systems offer point-in-time consistency of snapshots, which guarantees that all files are frozen in a state in which they were at the same point in time. However, this cannot be achieved in a distributed file system without global clocks or synchronous snapshot operations.

We present an algorithm for distributed file system snapshots that overcomes this problem by relaxing the point-in-time consistency of local file system snapshots to a time span-based consistency. Built on loosely synchronized server clocks, it makes snapshots available within milliseconds, without any kind of locking or synchronization. Our evaluation demonstrates that enabling and accessing snapshots involves a read/write throughput penalty of no more than 1% under normal conditions.

1. Introduction

For many of today's companies and organizations, stored data belongs to the most valuable assets. It is therefore essential to protect it from malicious manipulation, accidental deletion and hardware failures. Snapshotting and versioning file systems address these problems by retaining previous versions of files and directories, thus allowing users and administrators to perform roll-backs if necessary and to copy stable images to a backup system.

A wide range of snapshotting and versioning file systems have been developed throughout the last 25 years [1, 28, 2, 26, 33, 32, 15, 12, 5, 29, 10]. However, they were designed to run on local machines or single servers, while the need for storage space has outgrown the capacity of individual nodes. As a result, distributed storage has constantly gained in importance. In particular, the concept of object-based storage [8, 22] has emerged as the prevalent design pattern for modern distributed and parallel file systems, as object-based storage systems can be scaled out easily with inexpensive commodity hardware. Most of the top 500 supercomputers in the world use prominent representatives of object-based file systems like Lustre [6] or Panasas Active Scale [36] to accommodate their huge demands for storage resources. We believe that snapshotting mechanisms can protect large-scale object-based file systems from an unwanted loss or manipulation of data. Not only do they allow users to restore previous versions of their files, but they can also be used to create backups in a meaningful and well-defined state.

File system snapshots are generally expected to reflect all files and directories in a state in which they were at a certain point in time. We refer to this property as *point-in-time* consistency of snapshots, which is offered by a range of local file systems [1, 28, 15, 12, 5, 29]. In a distributed file system, however, enforcing this consistency semantics has major drawbacks. It would either require a synchronized snapshot operation that obviates changes to files while snapshots are being taken, or a global time. While the former has major performance implications, the latter cannot be put into practice, as clocks cannot be shared or perfectly synchronized between servers. However, it is feasible to synchronize server clocks in a loose manner, i.e. to effectively limit the drift of server clocks to an external reference clock. Technologies and protocols like GPS or NTP make it possible to set an upper bound on the clock drift in the range of milliseconds and less [30, 23].

We introduce the concept of *loose time synchrony* that relaxes point-in-time consistency of snapshots to a time span-based consistency. It ensures that the time span during which files on different servers are effectively captured in a snapshot is no larger than the upper bound on the clock drift between these servers. Backups based on such snapshots reflect all changes to all files within a narrow time frame, irrespective of the duration of the backup process or changes made to files and directories while the backup was created.

In this paper, we show that it is feasible to maintain loosely time-synchronized snapshots of a distributed object-based file system with a marginal performance penalty. We present a distributed snapshot algorithm that offers this semantics together with various other desirable features:

- New snapshots become available almost instantaneously. The time for taking a global snapshot of the entire file system can effectively be limited to milliseconds, regardless of the number of files and servers.
- Snapshots can be taken on-line, without affecting or inhibiting normal access to the file system.
- Single server failures have a limited impact on the availability of a snapshot, as only those parts that were stored on the affected servers become inaccessible.
- No server-to-server communication is needed. Since all communication is client-initiated, the algorithm scales to any number of servers.

The rest of the paper is structured as follows. Section 2 defines the concept of loose time synchrony. Section 3 outlines the principles of object-based file systems, for which section 4 presents our snapshot algorithm and section 5 discusses implementation-related issues. Section 6 presents an evaluation that points out the performance characteristics our our algorithm. Section 7 gives an overview of related work, followed by a conclusion in section 8.

2. Loose Time Synchrony

Generally speaking, a *consistent snapshot* is a stable image that reflects the state of a system at a certain point in time. Assuming that $\mathcal{S}_t$ is the state at the point in time t, a snapshot Σ_{t_0} taken at a point in time t_0 comprises the state $\mathcal{S}_{t_0}$, i.e. *latest* state $\mathcal{S}_t$ that existed *up to* the time t_0:

$$\begin{aligned}\Sigma_{t_0} &:= \{\mathcal{S}_{t_0}\} \\ &= \{\mathcal{S}_t : \nexists \mathcal{S}_{t'}(t < t' < t_0 \wedge \mathcal{S}_{t'} \neq \mathcal{S}_t) \wedge t \leq t_0\}\end{aligned} \quad (1)$$

To overcome the lack of a global time in a distributed system, we introduce the concept of *loose time synchrony*, a time-based consistency model for snapshots that accounts for drifting server clocks. We assume that each server $s \in S$ in the system has a local clock c_s, where $c_s(t)$ denotes the local time on s at the point in time t. These clocks shall proceed at approximately constant clock rates close to real time, as defined in the *timed asynchronous distributed system model* [7]. We further require these clocks to be loosely synchronized. This means that at any point in time t, an upper bound ϵ is known for the clock drift between any two servers s and s':

$$|c_s(t) - c_{s'}(t)| \leq \epsilon \quad (2)$$

A loosely time-synchronized snapshot $\Sigma^S_{t_0}$ taken at the point in time t_0 captures the local state of each server $s \in S$ at its local clock's projection of the point in time t_0, i.e. at its local time $c_s(t_0)$. $\Sigma^S_{t_0}$ is defined as follows, being $\mathcal{S}^s_t$ the local state of a server s at the point in time t:

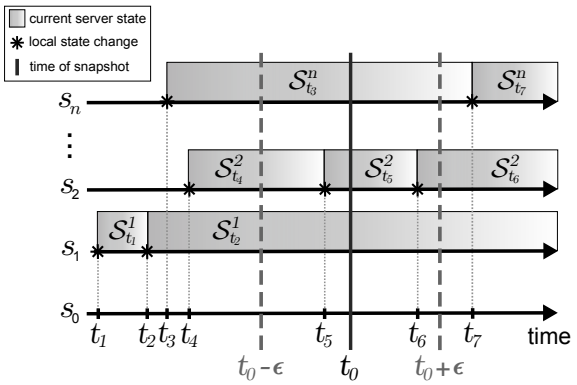


Figure 1: Illustration of loosely time-synchronized snapshot consistency.

$$\begin{aligned}\Sigma^S_{t_0} := \bigcup_{s \in S} \{\mathcal{S}^s_t : \nexists \mathcal{S}^s_{t'}(c_s(t) < c_s(t') < c_s(t_0) \wedge \mathcal{S}^s_{t'} \neq \mathcal{S}^s_t) \\ \wedge\ c_s(t) \leq c_s(t_0)\}\end{aligned} \quad (3)$$

Since server clock times may diverge by at most ϵ at any given point in time, $\Sigma^S_{t_0}$ is guaranteed to neither comprise any local server states $\mathcal{S}^s_{t'}$ for which a newer local state existed before $t_0 - \epsilon$, nor any local server states that started to exist after $t_0 + \epsilon$. Within the time frame $[t_0 - \epsilon, t_0 + \epsilon]$, ϵ sets an upper bound on the time span during which server states are effectively captured in a snapshot. Point-in-time snapshots can be regarded as a special case of loosely time-synchronized snapshots with $\epsilon = 0$.

Figure 1 illustrates the concept. Server s_0 takes a snapshot $\Sigma^S_{t_0}$ at the point in time t_0. On s_1, server state was changed to $\mathcal{S}^1_{t_1}$ at t_1 and to $\mathcal{S}^1_{t_2}$ at t_2, respectively. As both points in time were before $t_0 - \epsilon$, $\Sigma^S_{t_0}$ will contain $\mathcal{S}^1_{t_2}$ rather than $\mathcal{S}^1_{t_1}$. On s_2, any of the three states may be part of $\Sigma^S_{t_0}$, depending on s_2's clock c_2: if c_2 runs fast by more than $t_0 - t_5$ compared to c_0, its projection of t_0 will be in $[t_0 - \epsilon, t5]$, which causes $\mathcal{S}^2_{t_4}$ to be included; if it runs fast by at most $t_0 - t_5$ or slow by less than $t_6 - t_0$, $\mathcal{S}^3_{t_5}$ will be included accordingly; in any other case, $\mathcal{S}^2_{t_6}$ will be included. On s_n, $\mathcal{S}^n_{t_7}$ will never be included, regardless of c_n's drift, as the corresponding state change occurred after $t_0 + \epsilon$.

3. File System Architecture

The target environment for our snapshot algorithm are object-based [8, 22] file systems. Unlike traditional block-based file systems, they do not rely on a centralized management of disk blocks. Instead, individual services called *object storage devices (OSDs)* are responsible for the on-disk layout of file content. They store file content in the form of sequentially numbered byte ranges of the same size,

so-called *objects*. File system metadata, such as the directory tree, access rights or ownership information, is separately managed by a *metadata server (MDS)*. A client module translates requests to the file system into requests to the MDSs and OSDs. This design provides for a high degree of scalability, as a growing need for access bandwidth and storage capacity can be satisfied by adding new servers.

4. Object-based File System Snapshots

To implement loosely time-synchronized snapshots, OSDs and MDSs have to keep track of changes to their individual states and retain different versions of their data. Prior to a state change, a server needs to record the current local state and attach it to a new timestamped version.

The nature of a server's state depends on the server type. The state of an MDS is defined by its metadata, i.e. the set of its volumes, each consisting of a directory tree with the metadata of all its nested files, whereas the state of an OSD is defined by its file contents, i.e. the set of all objects that make up the data of its local files. Hence, we refer to a timestamped version of a server's state as a local *metadata snapshot* in case of an MDS and a local *file content snapshot* in case of an OSD, respectively.

State changes occur frequently across all servers and only affect small parts of the file system's state. For example, writing a file may cause a single object to be changed on an OSD and the file size to be updated on the MDS. Recording all state changes across all servers would lead to a huge number of local snapshots and thus a significant overhead in maintaining these. We therefore decided to track state changes (and thus take local snapshots) only in response to specific events.

On MDSs, such events are limited to explicit snapshot requests sent by users. On OSDs, we track `close` events of files rather than `write` events. Besides reducing the number of file content snapshots, this increases the chance that individual files are captured in meaningful states. If file content snapshots were taken regardless of whether files are currently open for writing, snapshotted files could be in partially written, unprocessable states. For similar reasons, various other versioning file systems also create file versions in response to `close` events [26, 33, 32, 21].

In the following, we describe how snapshots of server states are taken and how they are interrelated to implement loose time synchrony.

4.1. Metadata Snapshots

A metadata snapshot on an MDS provides the initial access point to a file system snapshot. Before file content can be read on an OSD, file metadata needs to be accessed on an MDS to determine the OSDs at which the file contents are stored.

In most cases, metadata servers only make up a small portion of all servers. To avoid bottlenecks in the metadata access path, it is of major importance that the overhead caused by taking metadata snapshots is kept as low as possible. We therefore assume that metadata servers are capable of taking point-in-time snapshots of their complete local metadata without severely affecting concurrent metadata accesses.

To maintain metadata snapshots, each MDS holds a snapshot table $T_M \subset \{(t, M)\}$, where t represents a timestamp and M a version of the local metadata. In response to a snapshot request, a new version M_{now} is captured, timestamped with the current time $c_{MDS}(t_{now})$ obtained from the MDS' local clock c_{MDS}, and added to T_M.

t_{now} acts as the timestamp for the corresponding global file system snapshot $\Sigma^{\mathcal{S}}_{t_{now}}$. To ensure that $\Sigma^{\mathcal{S}}_{t_{now}}$ is stable when being accessed, it is necessary to delay insertions in T_M by at least ϵ. Without this artificial delay, $\Sigma^{\mathcal{S}}_{t_{now}}$ could be accessed before the $t_{now} + \epsilon$, which might cause data from different file content snapshots to be read. That's because OSDs with clocks running slow compared to c_{MDS} may create new file content versions during the time interval $[t_{now}, t_{now} + \epsilon]$ that will be included in $\Sigma^{\mathcal{S}}_{t_{now}}$ according to the definition of loosely time-synchronized snapshots.

Figure 2 sketches the algorithm for metadata snapshots.

```
MDS::snapshot()
  capture current metadata version M_now
  t_s ← c_MDS(t_now)
  wait ε
  T_M ← T_M ∪ {(t_s, M_now)}
```

Figure 2: Management of metadata snapshots

4.2. File Content Snapshots

To provide the basis for file content snapshots, OSDs keep track of changes to their states at the granularity of a file's constituent objects. A simple way of doing this is to maintain versions of each object. Thus, each OSD holds a persistent set $O_f \subset \{(n, v, d)\}$ of versioned objects for each file f, where $n \in \mathbb{N}$ is the object number, $v \in \mathbb{N}$ is a version number for the object, and d points to the respective data on disk. We use copy-on-write (COW) techniques to preserve previous object versions when objects are written. Instead of overwriting the data d of an object (n, v, d), d is copied to a new data object d_{new} to which the changes δ are persistently applied (denoted as $d \circ \delta$). d_{new} is then linked to a new object $(n, v+1, d_{new})$ that is added to O_f. Thus, ob-

ject version v remains accessible in its original state while $v+1$ reflects the latest version.

File content snapshots are persistently stored in a snapshot table $T_f \subset \{(t, V) : V \subseteq O_f\}$ for each file f, where t refers to a timestamp and V to a file version, i.e. a set of object versions with different object numbers. In addition, each OSD keeps track of the latest version $V_f \subseteq O_f$ of each file f that reflects the file's current state.

Since we decided to take file content snapshots only in response to `close` events, it is not necessary to copy objects on every write. A new object version only needs to be created if the object has not been written since the file was opened for writing. To keep track of these objects, each OSD holds a transient set of object numbers $N_f \subset \mathbb{N}$ for each open file f. Depending on whether the object n to be written is already contained in N_f, the latest object version is either updated in place or copied prior to being written.

When a file is closed, the OSD creates a new file content snapshot by timestamping V_f with the current time $c_{OSD}(t_{now})$ on its clock and adding a corresponding snapshot table entry to T_f. Any transient open state of the file including N_f is discarded afterward.

Figure 3 shows the algorithm for file content snapshots.

```
OSD::write(f, n, δ)
  v_max ← max v : (n, v, d) ∈ O_f
  d_old ← d : {(n, v_max, d)} ∈ O_f
  d_new ← d_old ∘ δ
  if n ∈ N_f then
    O_f ← O_f ∪ {(n, v_max, d_new)} \ {(n, v_max, d_old)}
    V_f ← V_f ∪ {(n, v_max, d_new)} \ {(n, v_max, d_old)}
  else
    O_f ← O_f ∪ {(n, v_max + 1, d_new)}
    V_f ← V_f ∪ {(n, v_max + 1, d_new)} \ {(n, v_max, d_old)}
    N_f ← N_f ∪ {n}
  fi

OSD::close(f)
  T_f ← T_f ∪ {(c_OSD(t_now), V_f)}
  N_f ← ∅
```

Figure 3: Management of file content snapshots

4.3. Accessing File System Snapshots

We assume that files are accessed via path names. An MDS can resolve the path names for its locally stored volumes and retrieve the corresponding metadata, including the OSDs on which the file contents are stored. Once a file has been opened, its OSD is known to the client and can be accessed repeatedly until the file is closed.

Files in a snapshot are accessed in a similar way, except that a metadata snapshot is used by the MDS instead of the current version of the metadata. To specify a metadata snapshot, users provide an access timestamp t_a, which is sent to the MDS together with a client request. t_a defines an upper bound for the timestamp t_s assigned to the metadata snapshot that is supposed to be accessed. The MDS selects the most recent metadata snapshot (t_s, M) in T_M that was - according to its local clock - taken before t_a and executes the requested operation on M. Alternatively, snapshots could also be named when being created and retrieved by name when being accessed.

Reading content from a snapshotted file requires the correct file content version to be retrieved, subject to the definition of loose time synchrony. Thus, a client receives t_s from the MDS when opening a file and sends it together with the `read` request to the OSD that holds the file's content. To ensure loose time synchrony, the OSD selects the latest file content snapshot (t, V) in T_f that is not older than t_s according to its local clock and returns the data from the respective object in V. The fact that ϵ sets an upper bound on the asynchrony of server clocks ensures that $c_{OSD}(t) \in [t_s - \epsilon, t_s + \epsilon]$ if $t_s = c_{MDS}(t)$, which guarantees that V was neither outdated before $t - \epsilon$, nor created after $t + \epsilon$.

Figure 4 shows the algorithm for accessing snapshots.

```
MDS::select_snapshot(t_a)
  t_s ← max t : (t, M) ∈ T_M ∧ t ≤ c_MDS(t_a)
  return t_s

OSD::read(f, n, t_s)
  t_r ← max t : (t, V) ∈ T_f ∧ t ≤ t_s
  d_r ← d : (t_r, (n, v, d)) ∈ T_f
  return d_r
```

Figure 4: Reading data from a snapshot

5. Implementation

We implemented our snapshot algorithm in XtreemFS [14], a distributed object-based file system. An XtreemFS installation typically comprises a Directory Service (DIR) at which all services and volumes are registered, a Metadata and Replica Catalog (MRC) that acts as the MDS, as well as a set of OSDs.

The MRC is backed by BabuDB [34], a key-value store specifically designed for file system metadata management. BabuDB supports instantaneous snapshots without blocking concurrent access. Its architecture is based on concepts of LSM-trees [27] and Google Big Table [4]. Index structures are composed of an on-disk index and a stack of in-memory trees, on top of which a new tree is added each

time a snapshot is taken. Only the topmost tree may be changed, whereas the stack of trees below represents the latest immutable snapshot. This architecture makes it possible to take point-in-time snapshots of arbitrarily large indices without interrupting access to the database.

OSDs store file content in their local file systems. Each OSD maintains a directory per XtreemFS file, in which each object version is stored as a separate file. Object and version numbers are encoded in the file name. Snapshot table and current file version are also stored as files in the directory and updated when the file is closed and written, respectively. Before objects of a file can be accessed, the OSD checks if a transient open state exists for the file; if not, metadata such as the snapshot table T_f and the latest file version V_f are loaded into memory, where they reside until the file is closed. To ensure that files are closed despite client failures, there is no explicit close operation. Instead, files are implicitly closed if no keep-open message or request to access an object were received for a certain time span, which is usually no more than 60 seconds.

To allow for a clock synchronization between all OSDs and MRCs, the DIR also acts as a time server. Different mechanisms can be used to synchronize server clocks, including NTP and GPS.

5.1. File Size Consistency

The current size of a file is part of the file's metadata and needs to be returned by the MRC in response to a `stat` request. It is crucial that file sizes stored as part of the metadata are as accurate as possible, as many applications retrieve the current file size to determine the end of a file. XtreemFS uses an asynchronous protocol to ensure consistency of file content sizes and file sizes stored on the MRC [35]. When the file content size on an OSD changes in response to a `write` or `truncate` operation, the client receives the current file size from the OSD and reports it back to the MRC.

If a snapshot is taken before a file size update has been reported to the MRC, the file size stored with the metadata snapshot may be outdated. Likewise, snapshots taken between updating a file size at the MRC and closing the corresponding file at the OSD may cause wrong file sizes to be stored with the metadata. To circumvent the problem, it would be possible to fetch all file content sizes from all OSDs and attach the corrected ones to the metadata snapshot. As this would greatly reduce the speed and scalability of snapshot creation, however, we decided to modify the client-side implementation of `stat` instead, so that it fetches the correct file content size from the OSD instead of the MRC when invoked on a snapshot.

5.2. Cleanup

New file content snapshots are created on every `close` but never get deleted. Even though only changed objects of files require additional storage space, they will add up to a considerable size over time. However, many file content versions will never be accessed, as they are superseded by later versions that are created before a new metadata snapshot is taken.

We implemented a cleanup tool to dispose of obsolete file content versions, i.e. object versions that are not bound to any existing metadata snapshot or part of a file's current version. With each cleanup run, the tool first fetches the timestamps of all metadata snapshots from the MRC. Then, it triggers a cleanup operation on all OSDs with these timestamps. Each OSD compares the timestamps to the ones in the snapshot tables of its files, in order to discover those file content snapshots that are superseded. A file content snapshot (t, V) supersedes a file content snapshot (t', V') with a set of metadata snapshot timestamps T if the following condition holds true: $\nexists t_x \in T\ (t'-\epsilon < t_x < t+\epsilon)$. In this case, all object versions in V' may be deleted that are not contained in any other version V'' of a snapshot $(t'', V'') \in T_f$ or in V_f. As these object versions will never be accessed, their deletion may take place in a fully asynchronous manner.

6. Evaluation

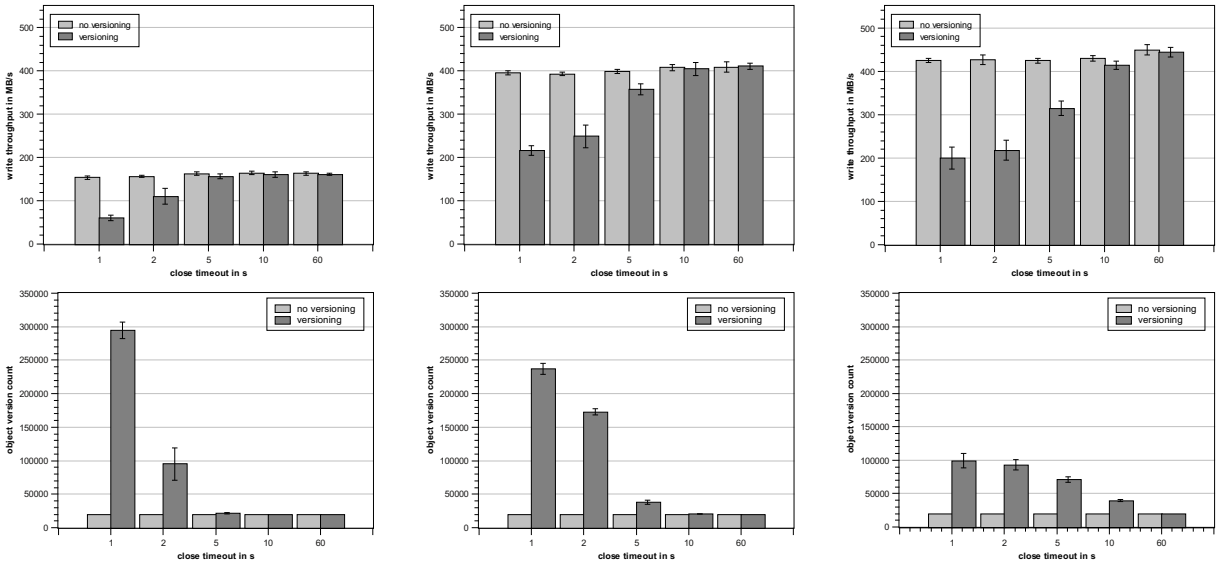
We conducted four experiments to demonstrate the characteristics of our snapshot algorithm and its implementation in XtreemFS:

- the impact of COW and versioning on write throughput,
- the impact of versioning and version lookups on snapshot read throughput,
- the additional latency of metadata operations experienced when accessing snapshots,
- the additional storage consumption induced by object versioning.

We used up to 16 nodes of a cluster connected by a gigabit Ethernet, each node being equipped with an 8-core 2.3 GHz CPU, 8 GB of RAM and a local hard drive. For the hard drives, `iostat` indicated I/O throughput rates of up to 90 MB/s.

6.1. Write Throughput

The first experiment points out the impact of file content versioning and COW on the write throughput. We set up an



(a) **object size: 4 kB** (b) **object size: 32 kB** (c) **object size: 128 kB**

Figure 5: Aggregated write throughput and object version count for object sizes of 4 kB, 32 kB, and 128 kB. The error bars show the standard deviation of the results of 10 test runs.

XtreemFS installation consisting of an MRC, 5 OSDs, and 10 multi-threaded clients, each running on a separate cluster node. To prepare the experiments, we created 20,000 files consisting of a single object and assigned each to a random OSD. We decided to only have one object per file in order to maximize the number of file versions created and hence the potential overhead caused by the versioning. Once all files were created, the 10 clients performed write operations for a duration of 60 seconds on randomly selected files. With each write, `object size - 1` bytes were written at offset `0`, so that the complete object except for the last byte was replaced. We kept the last byte to trigger the COW mechanism; otherwise, new versions would have been created without copying. We measured the aggregated number of write operations performed across all clients and counted the total number of object versions (i.e. file versions) created across all OSDs.

We conducted the experiment 10 times with varying object sizes, as well as varying timeouts after which files were closed on the OSDs and thus new versions were created. To quantify the cost of COW and versioning, we repeated the whole set of experiments twice, once with and once without versioning enabled. We used the average values of 10 test runs for our evaluation.

Figure 5 shows the number of operations per second and the number of file versions created for different object sizes and close timeouts. With versioning enabled, the diagrams illustrate that longer close timeouts generally lead to smaller numbers of versions and hence less COW operations, which becomes visible in growing throughput rates. With a standard close timeout of 60 seconds, the performance impact of COW becomes marginal (approx. 1%). However, even if almost 15 times the initial number of versions are created, the system can still sustain throughput rates of up to 40% of the maximum rates attained without versioning (see figure 5a, close timeout: 1s).

6.2. Read Throughput

The second experiment shows the impact of file content versioning on the read throughput. We used the same setup as for the write throughput measurements. We populated an empty file system with 20,000 object-size files, which we repeatedly overwrote in order to trigger the creation of new versions. After having overwritten each file 10 times and waited for the close timeouts between two overwrites in order to ensure that new versions were created, we triggered a snapshot and repeatedly read randomly-selected files in this snapshot for 60 seconds. We measured the aggregated number of read operations performed across all clients. We repeated this procedure 10 times, so that a total of 100 content versions per file were created. To compare our results, we also made test runs without versioning enabled. We conducted the experiment with varying object sizes and ran each test 10 times.

Figure 6 illustrates the outcome. With growing numbers of file versions, the graphs show a slight decrease in read throughput. However, even with an object size of 4kB and large numbers of snapshots, the decrease is only about 1%. The performance penalty is caused by growing file version

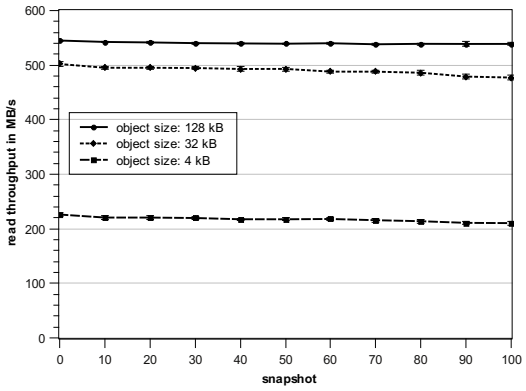


Figure 6: Aggregated read throughput for object sizes of 4 kB, 32 kB, and 128 kB. Snapshot 0 represents the first test run without versioning enabled, all other snapshots are numbered according to their chronological order. The error bars show the standard deviation of the results of 10 test runs.

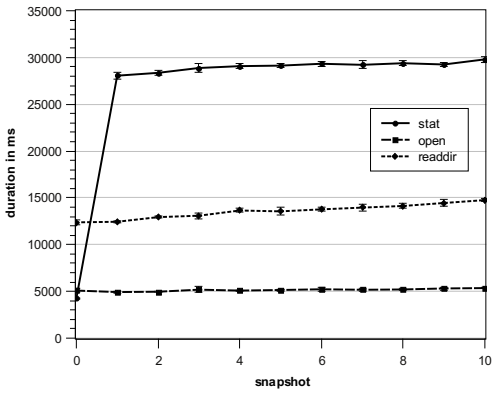


Figure 7: Access latency of metadata snapshots. Snapshot 0 represents the first test run without snapshots, all other snapshots are numbered according to their chronological order. The error bars show the standard deviation of the results of 10 test runs.

tables as well as numbers of objects per file, which leads to longer version table load and object version lookup times.

Note that the total read throughput attained with large object sizes is bound by the network and thus higher than the accumulated disk I/O maximum across all OSDs of approximately 450 MB/s. This is because many read requests did not hit the OSDs' hard disks but were served from the cache instead.

6.3. Metadata Access Latency

The third experiment quantifies the additional latency experienced when accessing snapshots, which originates from the maintenance overhead of file and metadata versions. We set up an XtreemFS installation with one MRC and OSD and one single-threaded client on different cluster nodes. To assess the latency impact of snapshots on metadata accesses, we populated a volume with 10,000 files, which we randomly distributed across a tree of 500 directories with an average depth of 4. We initially performed a test run before snapshots were created, so as to be able to compare our results. Then, we repeated the procedure of taking a snapshot, performing a test run on this snapshot, deleting the tree, and creating the same tree again 10 times. With each test run, we measured the average duration of 10,000 *open*, *readdir* and *stat* operations.

The results are shown in figure 7. While the graphs suggest that `open` has an almost negligible latency impact when being invoked on a snapshot, `readdir` shows a slightly increased latency with a growing number of snapshots. This comes from the MRC's internal database design: the larger the number of snapshots, the more index structures need to be merged to perform a `readdir` operation [34]. The `stat` operation shows a substantially lower latency without versioning, as no additional file size glimpse on the OSD is needed.

6.4. Storage Consumption

The fourth experiment aims to determine the additional storage consumption that is caused by file content versioning. We set up a complete XtreemFS installation on a single node. We executed the `Postmark` file system benchmark in order to generates a randomized mixed write/append workload. We configured `Postmark` to initially create 1,000 files of an evenly distributed random size between 512 bytes and 1 MB, and to subsequently modify these files by performing 10,000 random size append writes until the maximum file size of 1 MB was reached. After having run the benchmark, we measured the total amount of disk space occupied by all files on the OSD, purged all previously created versions by running our cleanup tool, and measured the occupied disk space again. We repeatedly conducted the experiment with different object sizes and repeated each individual test run 10 times. We set the close timeout to one second, so as to ensure that as many file versions as possible were created.

Figure 8 shows the results for different object sizes of up to 1,024 kB. Without cleaning up, larger object sizes lead to a higher storage consumption, as the size of the data chunks that need to be copied on write directly correlates with the object size. While an object size of 1,024 kB leads to a significant amount of data of more than 7.3 times the original volume, object sizes of up to 128 kB limit the amount to a factor of 1.6 and less. The final cleanup run removes all

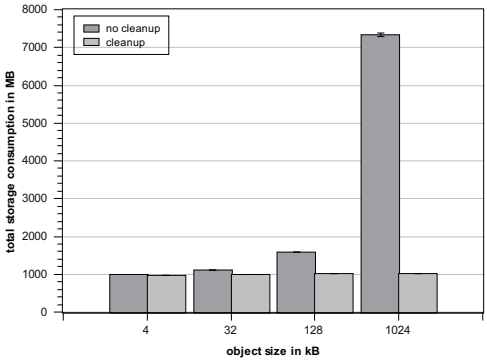


Figure 8: Total storage consumption after `Postmark` test runs with different object sizes. The error bars show the standard deviation of the results of 10 test runs.

file content versions but the latest one, as no metadata snapshots were taken while running the experiments. Thus, the total data volume can be substantially lowered by selecting appropriate object sizes and purging obsolete file content versions on a regular basis.

7. Related Work

Most related work on file system snapshots refers to local file systems. Various snapshotting file systems have been developed throughout the last decades, such as ZFS [1], ext3cow [28], WAFL [12], Episode [5], Spiralog [15] and Plan-9 [29]. Most of them are block-based and use COW techniques at inode and block level to retain prior versions of files and directories. Versions are linked to snapshots by means of locally assigned timestamps. Since all timestamps are assigned by the same clock, point-in-time snapshots can be created without the problem of clock synchrony.

Some distributed file systems support snapshots. Spiralog [15] is a network file system built upon the log-structured file system LFS [31]. Rather than maintaining files and directories at fixed locations on disk, each LFS-based server uses a log to record all changes. To take a point-in-time snapshot, it is sufficient to adequately mark the last log entry. However, the scale of a snapshot is limited to a single server, as each server holds its own log. Similarly, AFS [13] provides snapshotting functionality at volume granularity but does not allow volumes to be distributed across multiple servers. Frangipani [37], a distributed file system built on top of the Petal virtual disk system [17] also supports snapshots but requires file system activity to be interrupted during the process of taking a snapshot. The Google File System [9] supports snapshots at a large scale but does not guarantee point-in-time semantics or upper bounds on the snapshot time span across multiple servers.

The idea of using loosely synchronized clocks to implement distributed consistent snapshots has been described by Moh and Liskov in the context of TimeLine [24], a distributed object-oriented database system. The algorithms and concepts of TimeLine resemble the ones described in this paper, but the main focus is on transactions. This leads to a more complex less scalable snapshot protocol, as it relies on communication between servers to disseminate information about snapshots.

An alternative approach to overcome the problem of global time is to relax point-in-time consistency of snapshots to a causality-based consistency model. A snapshot is *causally consistent*, also referred to as a *consistent cut* [20], if it reflects the causal dependencies between all states contained. Causal dependencies are generally defined via messages exchanged between servers; a causally consistent snapshot that comprises the state of a server B after having received a message m from a server A must also comprise a state in which A was after it sent m to B. Chandy and Lamport introduced an algorithm that captures a causally consistent snapshot of a distributed system with FIFO message delivery guarantees [3]. Lai and Yang [16], Li et. al. [18], Helary [11] and Mattern [19, 20] introduced alternative algorithms that do not have this FIFO restriction. However, such algorithms are difficult to implement in object-based file systems. Their system model corresponds to a fully connected graph of equivalent peers, whereas object-based file systems rely on a client-server model with stateless and only temporarily connected clients. This hinders the propagation of snapshot-related information.

Moreover, causal dependencies that originate from communication outside the file system cannot be tracked easily. For a local file system, Muniswamy-Reddy and Holland solved this problem by monitoring all local communication channels (including sockets, pipes, etc.) at operating system level [25]. In an object-based file system, however, such an approach would require a distributed monitoring infrastructure that spans all clients and servers, which would greatly increase the complexity of the system and limit the applicability of the approach.

8. Conclusion

We have presented a snapshot algorithm for large-scale distributed object-based file systems. The algorithm guarantees loose time synchrony of snapshots, a time-based consistency model that relaxes point-in-time constraints of local snapshotting file systems to time span-based constraints. Provided that an upper bound ϵ on the drift between all server clocks is known, the algorithm makes it possible to capture a stable image of a file system within a time span of no more than ϵ, which can be limited to milliseconds with protocols like NTP or GPS-synchronized clocks. All files

in a snapshot are guaranteed to reflect states in which they were no more than ϵ before the snapshot was initiated. The algorithm is agnostic to the scale of the file system in terms of the number of servers, as it does not rely on communication between servers. Taking a snapshot is a non-inhibitory metadata operation that has only a marginal impact on concurrent metadata accesses. Our experiments have demonstrated that versioning causes a loss in read and write performance of less than 1% under normal conditions.

Acknowledgment

This work was supported by the EU IST program as part of the XtreemOS project (contract FP6-033576) and the Contrail project (contract FP7-257438).

References

[1] J. Bonwick and B. Moore. ZFS: The last word in file systems. http://opensolaris.org/os/community/zfs/docs/zfs last.pdf, 2007.

[2] F. Bustamante, B. Cornell, B. Cornell, P. Dinda, P. Dinda, and F. Bustamante. Wayback: A user-level versioning file system for Linux. In *Proceedings of USENIX 2004 (Freenix Track)*, 2004.

[3] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[4] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. In *OSDI '06: Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 205–218, 2006.

[5] S. Chutani, O. T. Anderson, M. L. Kazar, B. W. Leverett, W. A. Mason, R. N. Sidebotham, and T. Corporation. The Episode file system. In *Proceedings of the USENIX Winter 1992 Technical Conference*, pages 43–60, 1992.

[6] Cluster File Systems, Inc. Lustre: a scalable, high-performance file system. Lustre Whitepaper Version 1.0, 2002.

[7] F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, 10:642–657, 1999.

[8] M. Factor, K. Meth, D. Naor, O. Rodeh, and J. Satran. Object storage: The future building block for storage systems. In *2nd International IEEE Symposium on Mass Storage Systems and Technologies*, pages 119–123, 2005.

[9] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *SOSP '03: Proceedings of the nineteenth ACM Symposium on Operating Systems Principles*, pages 29–43, 2003.

[10] D. K. Gifford, R. M. Needham, and M. D. Schroeder. The Cedar file system. *Commun. ACM*, 31(3):288–298, 1988.

[11] J.-M. Hélary. Observing global states of asynchronous distributed applications. In *Proceedings of the 3rd International Workshop on Distributed Algorithms*, pages 124–135, London, UK, 1989. Springer-Verlag.

[12] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 235–246, Berkeley, CA, USA, 1994. USENIX Association.

[13] J. H. Howard, M. L. Kazar, S. G. Menees, A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6:51–81, 1988.

[14] F. Hupfeld, T. Cortes, B. Kolbeck, J. Stender, E. Focht, M. Hess, J. Malo, J. Marti, and E. Cesario. XtreemFS: a case for object-based storage in grid data management. In *3rd VLDB Workshop on Data Management in Grids*, 2007.

[15] J. E. Johnson and W. A. Laing. Overview of the Spiralog file system. *Digital Tech. J.*, 8(2):5–14, 1996.

[16] T. H. Lai and T. H. Yang. On distributed snapshots. *Information Processing Letters*, 25(3):153–158, 1987.

[17] E. K. Lee, C. A. Thekkath, and R. A. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 84–92, 1996.

[18] H. F. Li, T. Radhakrishnan, and K. Venkatesh. Global state detection in non-fifo networks. In *ICDCS*, pages 364–370, 1987.

[19] F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, pages 215–226. North-Holland, 1989.

[20] F. Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing*, 18(4):423–434, 1993.

[21] K. McCoy. *VMS File System Internals*. Digital Press, 1990.

[22] M. Mesnier, G. Ganger, and E. Riedel. Object-based storage. *IEEE Communications Magazine*, 8:84–90, 2003.

[23] D. L. Mills. A brief history of NTP time: Memoirs of an internet timekeeper. *SIGCOMM Computer Communication Review*, 33(2):9–21, 2003.

[24] C.-H. Moh and B. Liskov. TimeLine: A high performance archive for a distributed object store. In *1st Symposium on Networked Systems Design and Implementation (NSDI 2004)*, pages 351–364, 2004.

[25] K.-K. Muniswamy-Reddy and D. A. Holland. Causality-based versioning. In *FAST '09: Proceedings of the 7th USENIX Conference on File and Storage Technologies*, pages 15–28, 2009.

[26] K.-K. Muniswamy-Reddy, C. P. Wright, A. Himmer, and E. Zadok. A versatile and user-oriented versioning file system. In *FAST '04: Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 115–128, Berkeley, CA, USA, 2004. USENIX Association.

[27] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Inf.*, 33(4):351–385, 1996.

[28] Z. Peterson and R. Burns. Ext3cow: A time-shifting file system for regulatory compliance. *ACM Transactions on Storage*, 1(2):190–212, 2005.

[29] S. Quinlan. A cached WORM file system. *Software Practice and Experience*, 21(12):1289–1299, 1991.

[30] J. Ridoux and D. Veitch. Principles of robust timing over the internet. *Communications of the ACM*, 53(5):54–61, 2010.

[31] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10:1–15, 1991.

[32] D. Santry, M. J. Feeley, N. C. Hutchinson, and A. C. Veitch. Elephant: The file system that never forgets. In *Workshop on Hot Topics in Operating Systems*, pages 2–7. IEEE Computer Society, 1999.

[33] C. A. N. Soules, G. R. Goodson, J. D. Strunk, and G. R. Ganger. Metadata efficiency in versioning file systems. In *FAST '03: Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, 2003.

[34] J. Stender, B. Kolbeck, M. Högqvist, and F. Hupfeld. BabuDB: Fast and efficient file system metadata storage. In *6th IEEE International Workshop on Storage Network Architecture and Parallel I/Os (SNAPI)*, 2010.

[35] J. Stender, B. Kolbeck, F. Hupfeld, E. Cesario, E. Focht, M. Hess, J. Malo, and J. Marti. Striping without sacrifices: Maintaining POSIX semantics in a parallel file system. In *LASCO'08: First USENIX Workshop on Large-Scale Computing*, 2008.

[36] H. Tang, A. Gulbeden, J. Zhou, W. Strathearn, T. Yang, and L. Chu. The Panasas ActiveScale storage cluster - delivering scalable high bandwidth storage. In *Proceedings of the ACM/IEEE SC2004 Conference (SC'04)*, page 53, November 2004.

[37] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A scalable distributed file system. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 224–237, 1997.