A Transparently-Scalable Metadata Service for the Ursa Minor Storage System

Shafeeq Sinnamohideen[†] Raja R. Sambasivan[†] James Hendricks[†⋆]
Likun Liu[‡] Gregory R. Ganger[†]
[†]*Carnegie Mellon University* [‡]*Tsinghua University* [⋆]*Google*

Abstract

The metadata service of the Ursa Minor distributed storage system scales metadata throughput as metadata servers are added. While doing so, it correctly handles metadata operations that involve items served by different metadata servers, consistently and atomically updating the items. Unlike previous systems, it does so by reusing existing metadata migration functionality to avoid complex distributed transaction protocols. It also assigns item IDs to minimize the occurrence of multi-server operations. Ursa Minor's approach allows one to implement a desired feature with less complexity than alternative methods and with minimal performance penalty (under 1% in non-pathological cases).

1 Introduction

Ursa Minor is a scalable storage system designed to support automation [1]. It, like other direct-access storage systems [14], is structured in two primary parts: a data path for handling data access and a metadata path for handling metadata access. This separation allows each path to be optimized for its purpose. Modern scalable storage systems are expected to scale to thousands of storage nodes and tens or hundreds of metadata nodes, so each part is a distributed system in its own right. This paper explains the goals for the metadata path of Ursa Minor, describes the design and implementation of a prototype that fulfills them, and introduces and evaluates a novel technique for handling multi-server operations simply and efficiently.

As a whole, an Ursa Minor *constellation*, consisting of data nodes and metadata nodes, is expected to be highly available and durable, like other distributed storage systems. Ursa Minor is also intended to be incrementally scalable, allowing nodes to be added to or removed from the system as storage requirements change or hardware replacement becomes necessary. To provide for this, Ursa Minor must include a mechanism for *migrating* data from one data node to another and metadata from one metadata node to another.

The overall goals for Ursa Minor demand that the metadata path be *transparently scalable*. That is, it should be able to scale in capacity and throughput as more nodes are added, while users and client applications should not have to be aware of the actions the system takes to ensure scalability — the visible semantics should be consistent regardless of how the system chooses to distribute metadata across metadata nodes. Several existing systems have demonstrated this design goal (e.g., [3, 6, 36]).

The *scalable* part of the requirement implies that the system will use multiple metadata nodes, with each storing some subset of the metadata and servicing some subset of the metadata requests. In any system with multiple servers, it is possible for the load across servers to be unbalanced; therefore, some mechanism for load balancing is desired. This can be satisfied by migrating some of the metadata from an overloaded server to a less loaded one, thus relocating requests that pertain to that metadata.

The *transparent* part implies that clients see the same behavior regardless of how the system has distributed metadata across servers. Any operation on any single object should have the same result, no matter which server happens to be responsible for that object at that time. The same should hold for operations that involve more than one object, even if the objects involved are distributed to different servers, and even if some of those servers fail during the operation. This is an area that many existing systems leave unfinished for future work. Ursa Minor correctly and efficiently handles multi-server operations in a novel manner. The approach taken by Ursa Minor reuses the mechanism for migrating object metadata to implement multi-server operations by migrating the required metadata to a single metadata server and executing the operation on that server. This is a key contribution of our work because it simplifies the implementation of the metadata path while still providing good performance and the same failure semantics as a single server. To prevent most multi-file operations from being multi-server operations, Ursa Minor uses an object ID assignment policy that translates namespace locality into object ID locality.

Experimental results show that Ursa Minor scales linearly in metadata throughput when executing a variant of

the SPECsfs97 benchmark. On workloads that contain frequent multi-server operations, performance degrades in proportion to the frequency of multi-server operations. This slowdown is due to the overhead of migration and is only 1.5% for workloads where multi-server operations are 10× more frequent than the worst-case behavior inferred from traces of deployed file systems.

The remainder of this paper is organized as follows: Section 2 reviews scalable distributed file systems, cross-server operations, and other related work. Section 3 describes the design of Ursa Minor's metadata path. Section 4 describes our evaluation of the prototype's performance and the trace analysis we performed in order to characterize expected workloads.

2 Background

Many distributed file systems have been proposed and implemented over the years. Architects usually aim to scale the capacity and throughput of their systems by doing one or more of the following:

- Increasing the capability of individual servers.
- Reducing the work each server performs per client.
- Increasing the number of servers in the system.

Each of these axes is largely independent of the others. As a research platform for exploring the issues of scaling distributed storage systems, Ursa Minor currently focuses on the last approach. Any work done here will still apply as the capability of each individual server improves. Such improvements would, of course, increase the capability of the entire constellation. Most existing work on decreasing a server's per-client workload focuses on the client-server protocols [18, 26, 32]. Historically, the adoption of improved protocols has been slowed by the need to modify every client system to use the new protocol. Recently, however, some of these techniques have been incorporated into the NFSv4 standard that is expected to be widely adopted [33]. Like the SpinFS protocol [11], Ursa Minor's internal protocol is designed to efficiently support the semantics needed by CIFS [27], AFS [18], NFSv4, and NFSv3 [8]. At present, however, we have only implemented the subset needed to support NFSv3.

As mentioned previously, challenges in scaling the number of servers in a system include handling the infrequent operations that involve multiple servers and managing the distribution of files across servers. The remainder of this section discusses operations that could involve multiple servers, how close existing systems come to being transparently scalable, how systems that handle multi-server operations transparently do so, and the importance of migration in a multi-server file system.

2.1 Multi-item operations

There are a variety of file system operations that manipulate multiple files, creating a consistency challenge when the files are not all on the same server. Naturally, every CREATE and DELETE involves two files: the parent directory and the file being created or deleted. Most systems, however, assign a file to the server that owns its parent directory. At some points in the namespace, of course, a directory must be assigned somewhere other than the home of its parent. Otherwise all metadata will be managed by a single metadata server. Therefore, the CREATE and DELETE of that directory will involve more than one server, but none of the other operations on it will do so. This section describes other significant sources of multi-item operations.

The most commonly noted multi-item operation is RENAME, which changes the name of a file. The new name can be in a different directory, which would make the RENAME operation involve both the source and destination parent directories. Also, a RENAME operation can involve additional files if the destination name exists (and thus should be deleted) or if the file being renamed is a directory (in which case, the '..' entry must be modified and the path between source and destination traversed to ensure a directory will not become a child of itself). Application programming is simplest when the RENAME operation is atomic, and both the POSIX and the NFSv3 specifications call for atomicity.

Many applications rely on this specified atomicity as a building-block to provide application-level guarantees. For example, many document editing programs implement atomic updates by writing the new document version into a temporary file and then using RENAME to move it to the user-assigned name. Similarly, many email systems write incoming messages to files in a temporary directory and then RENAME them into a user's mailbox directory. Without atomicity, applications and users can see strange intermediate states, such as two identical files (one with each name) existing or one file with both names as hard links.

Creation and deletion of hard links (LINK and UNLINK) are also multi-item operations in the same way that CREATE is. However, the directory the link is to be created in may not be the parent of the the file being linked to, making it more likely that the two are on different servers than for a CREATE and UNLINK.

The previous examples assume that each directory is indivisible. But a single heavily used directory might have more traffic than a single server can support. Some systems resolve this issue by splitting directories and assigning each part of the directory to a different server [37, 29]. In that case, simply listing the entire directory requires an operation on every server across which it is

split, and renaming a file within a directory might require two servers if the source name is in one part of the directory and the destination is in a different part.

Transactions are a very useful building block. Modern file systems, such as NTFS [28] and Reiser4 [31], are adding support for multi-request transactions. For example, an application could update a set of files atomically, rather than one at a time, and thereby preclude others seeing intermediate forms of the set. This is particularly useful for program installation and upgrade. The files involved in such a transaction could very easily be spread across servers.

Point-in-time snapshots [9, 17, 25, 30] have become a mandatory feature of most storage systems, as a tool for consistent backups, on-line integrity checking [25], and remote mirroring of data [30]. Snapshot is usually supported only for entire file system volumes, but some systems allow snapshots of particular subtrees of the directory hierarchy. In any case, it is clearly a substantial multi-item operation, with the expectation that the snapshot captures all covered files at a single point in time.

2.2 Transparent scalability

We categorize existing systems into three groups based on how fully they provide transparent scalability as the number of servers increases. Transparent scaling implies scaling without client applications having to be aware of how data is spread across servers; a distributed file system is not transparently scalable if client applications must be aware of capacity exhaustion of a single server or different semantics depending upon which servers hold accessed files.

No transparent scalability: Many distributed file systems, including those most widely deployed, do not scale transparently. NFS, CIFS, and AFS all have the property that file servers can be added, but each serves independent file systems (called *volumes*, in the case of AFS). A client can mount file systems from multiple file servers, but must cope with each server's limited capacity and the fact that multi-file operations (e.g., RENAME) are not atomic across servers.

Transparent data scalability: An increasingly popular design principle is to separate metadata management (e.g., directories, quotas, data locations) from data storage [6, 13, 14, 36, 38]. The latter can be transparently scaled relatively easily, assuming all multi-object operations are handled by the metadata servers, since each data access is independent of the others. Clients interact with the metadata server for metadata activity and to discover the locations of data. They then access data directly at the appropriate data servers. Metadata semantics and policy management stay with the metadata server, permitting simple, centralized solutions. The metadata server can limit throughput, of course, but off-loading data accesses pushes the overall system's limit much higher [14]. To go beyond this point, the metadata service must also be scalable.

Most modern storage systems designed to be scalable fall into this category. Most are implemented initially with a single metadata server, for simplicity. Examples include Google FS [13], NASD [14], Panasas [38], Lustre [24], prior versions of Ursa Minor [1], and most SAN file systems. These systems are frequently extended to support multiple metadata servers, each exporting a distinct portion of the namespace, and the ability to dynamically migrate files from one metadata server to another. Such a solution, however, is not transparently scalable because clients see different semantics for operations that cross metadata server boundaries.

Full transparent scalability: A few distributed file systems offer full transparent scalability, including Farsite [10], GPFS [32], Frangipani [36], and the version of Ursa Minor described in this paper. Most use the data scaling architecture above, separating data storage from metadata management. Then, they add protocols for handling metadata operations that span metadata servers. Section 2.3 discusses these further.

Another way to achieve transparent scalability is to use a virtualization appliance or "file switch" with a collection of independent NFS or CIFS file servers [7, 19, 20, 39]. The file switch aggregates an ensemble of file servers into a single virtual server by interposing on and redirecting client requests appropriately. In the case of multi-server operations, the file switch serves as a central point for serialized processing and consistency maintenance, much as a disk array controller does for a collection of disks. Thus, the virtual server remains a centralized, but much more capable, file system.

2.3 Multi-server operations

Traditionally, multi-server operations are implemented using a distributed transaction protocol, such as a two-phase commit [15]. Since each server already must implement atomic single-server operations, usually by using write-ahead logging and roll-back, the distributed transaction system can be built on top of the local transaction system. A transaction affecting multiple servers first selects one to act as a coordinator. The coordinator instructs each server to add a PREPARE entry, covering that server's updates, to their local logs. If all servers PREPARE successfully, the transaction is finalized with a COMMIT entry to all logs. If the PREPARE did not succeed on all servers, the coordinator instructs each server to roll back its state to the beginning of the transaction. With single-server transactions, recovering from a crash requires a server to examine its log and undo any incomplete transactions. Recovery from a multi-server transaction, however, is much more complicated.

With more than one server, it is possible for some servers to crash and others survive. If one crashed during PREPARE, the coordinator will wait until a time-out, then instruct the other servers to roll back their PREPAREs. If one crashed between PREPARE and COMMIT, when that server restarts, it needs to discover whether it missed the instruction to either COMMIT or UNDO. To do so, it needs to contact the coordinator or the other servers to determine whether any of them committed (in which case the coordinator must have successfully PREPAREd at all servers). Any step involving communication with other servers may fail, and if other servers have crashed, it may not be possible to proceed until they are online.

Distributed transactions may complicate other aspects of the system as well. Concurrency control within a single server requires each transaction to acquire locks to protect any state it operates on. The same is true for a multi-server operation, but now it is possible for the lock holder to crash independently of the server managing the lock. While there are existing techniques, such as leases, to handle this situation, a lock recovery scheme is simply not needed when locks can only be local to a server. Considering other common faults, such as an intermittent network failure, adds even more cases to handle.

As discussed, most of the additional complexity is in the recovery path. Not only must the recovery path handle recovery from a wide variety errors or crashes, it must also handle errors during recovery. This leads to a large number of cases that must be detected and handled correctly. Since errors in general are rare, and any particular error is even rarer, bugs in the fault-handling path may be triggered rarely and be even harder to reproduce. This places more reliance on test harnesses, which must be crafted to exercise each of the many error conditions and combinations thereof.

In order to minimize the rarely-used additional complexity of distributed transactions, Ursa Minor takes a novel approach to implementing multi-server operations. When a multi-server operation is required, the system migrates objects so that all of the objects involved in an operation are assigned to the same server, and the operation is then performed locally on that single server. This scheme is discussed in more detail in Section 3.5 and requires only single-server transactions and migration. Migration is itself a simplified distributed transaction, but it must already be implemented in the system to provide even non-transparent scalability.

2.4 Migration

In any system with many metadata servers, the question arises as to which files should be assigned to which servers. Some systems, such as AFS, NFS, Panasas, and Lustre, split the file system namespace into several *volumes* and assign each metadata server one or more volumes whose boundaries cannot be changed after creation. Others, such as xFS, Ceph, and OntapGX, are able to assign individual files to distinct servers. In general, supporting finer granularities requires more complexity in the mechanism that maps files to metadata servers.

Managing large-scale storage systems would be very difficult without migration — at the very least, hardware replacement and growth must be accounted for. Additionally, migration is a useful tool for addressing load or capacity imbalances. Almost every storage system has some way of performing migration, in the worst case by backing up data on the original server, deleting it, and restoring on the destination server.

Such offline migration, however, is obtrusive to clients, which will notice periods of data unavailability. If the need for migration is rare, it can be scheduled to happen during announced maintenance periods. As a system gets larger, the need for migration increases, while the tolerance for outages decreases. To address this issue, many modern systems [11, 18, 37, 38] can perform migration dynamically, while serving client requests, leaving clients unaffected except for very brief periods of unavailability. Any such system would be able to utilize the same approach used in Ursa Minor to provide transparent scalability.

The process of assigning files to servers can be thought of as analogous to lock management. A server that is assigned responsibility for a file (or collection of files) has effectively been granted an exclusive lock on that file and migration changes the ownership of that lock. Given the relative rarity and granularity of migration, the centralized migration managers used in AFS [18] and OntapGX [11] need not be as efficient or complex as the distributed lock managers used for fine-grained locking in GPFS [32], Slice [5], and Frangipani [36].

3 Design

Ursa Minor is a scalable storage system, designed to scale to thousands of storage nodes. Ursa Minor is a direct-access storage system [14], consisting of storage nodes and metadata servers. The storage nodes, termed *workers*, store byte streams named by Object ID, termed *SOID*[1]. There are no restrictions on which objects can reside on which worker, and an object's data can be replicated or erasure-coded across multiple workers, allowing the flexibility to tune an individual object's level of fault-tolerance and performance to its particular needs. Accessing a particular file's data requires two steps: first, the file name must be translated to a SOID, and second, the workers(s) responsible for the file data must be identified so that they can be contacted to retrieve the data. In Ursa Minor, these functions are performed by

[1] A Self-* Object ID is a 128 bit number analogous to an inode number and unique across an Ursa Minor constellation

the Namespace Service (NSS) and the Metadata Service (MDS), respectively. This section describes the high-level organization of these services and provides more detail on the internal components that enable transparent scalability.

3.1 Metadata Service (MDS)

The Metadata Service in Ursa Minor maintains information on each object, similar to that maintained by the inodes of a local-disk file system. For each object, the MDS maintains a record that includes the object's size, link count, attributes, permissions, and the list of worker(s) storing its data. Clients communicate with the MDS via RPCs. Since clients are untrusted, the MDS must verify that each request will result in a valid state and that the client is permitted to perform that action. Some requests, such as creating or deleting an object, require the MDS to coordinate with workers. Others, such as updating an attribute or timestamp, reside wholly within the MDS. The semantics defined for the MDS imply that individual requests are atomic (they either complete or they don't), consistent (the metadata transitions from one consistent state to another), independent (simultaneous requests are equivalent to some sequential order), and durable (once completed, the operation's results will never be rolled back). The transaction mechanism used to ensure this is discussed in detail in Section 3.6.

The MDS is responsible for all object metadata in Ursa Minor. Individual object metadata records are stored in metadata *tables*. Each table includes all records within a defined range of SOIDs. The tables are internally structured as B-trees indexed by SOID and are stored as individual objects within Ursa Minor. The ranges can be altered dynamically, with a minimum size of one SOID, and a maximum of all possible SOIDS. Within those limits, the MDS may use any number of tables, and, collectively, the set of tables contains the metadata for all objects. Storing the tables as objects in Ursa Minor allows the MDS to benefit from the reliability and flexibility provided by Ursa Minor's data path, and results in the metadata path holding no hard system state.

Each Ursa Minor cluster includes one or more metadata servers. Each metadata server is assigned a number of metadata tables, and each table is assigned to at most one server at a time. Thus, accessing the metadata of any particular object will only involve one server at a time. Because the metadata tables are themselves objects, they can be accessed by any metadata server using Ursa Minor's normal data I/O facilities.

The assignment of tables to servers is recorded in a *Delegation Map* that is persistently maintained by a *Delegation Coordinator*. The delegation coordinator is co-located with one metadata server, termed the *Root Metadata Server*. This server is just like any other metadata server, except it happens to host the metadata for the objects used by the metadata service. Clients request the delegation map when they want to access an object for which they do not know which metadata server to contact. They cache the delegation map locally and invalidate their cached copy when following a stale cached delegation map results in contacting the wrong metadata server. Tables can be reassigned from one server to another dynamically by the delegation coordinator, and this process is discussed in Section 3.4.

3.2 Namespace Service (NSS)

The Namespace Service manages directory contents. Directories are optional in Ursa Minor — applications satisfied with the MDS's flat SOID namespace (e.g.: databases, mail servers, scientific applications) need not use directories at all. Other applications expect a traditional hierarchical directory tree, which the Namespace Service provides.

Similarly to a local-disk file system, a directory entry is a record that maps a filename to a SOID. Directories are B-tree structured, indexed by name, and stored as ordinary objects, with their own SOIDs. At present, each directory object contains all of the directory entries for that directory, though there is no obstacle to splitting a directory across multiple objects.

Namespace servers are tightly coupled with metadata servers (in our implementation, both are combined in one server process which exports both RPC interfaces). Each namespace server is responsible for directories whose SOIDs are within the range exported by its coupled metadata server. This ensures that a directory's "inode" (the attributes stored by the MDS) and its contents will always be served by the same process. For the rest of this paper, we use the term "metadata server" to refer to the combined MDS and NSS server.

The NSS aims to support directory semantics sufficient to implement an overlying file system with POSIX, NFS, CIFS, or AFS semantics. As such, it provides the POSIX notions of hard links, including decoupling of unlink and deletion, and the ability to select how already-existing names are handled. Typical operations include creating a file with a given name, linking an existing object under a new name, unlinking an file, looking up the SOID corresponding to a file name, and enumerating the contents of directories.

3.3 SOID assignment

In Ursa Minor, the SOID of an object determines which table, and thus which metadata server, that object is assigned to. It follows that there may be advantages in choosing to use particular SOIDs for particular files. For instance, the **ls -al** command will result in a series of

requests, in sequential order, for the attributes of every file in a given directory. If those files all had numerically similar SOIDs, their metadata would reside in the same (or nearby) B-tree pages, making efficient use of the server's page cache. Similarly, most file systems exhibit spatial locality, so an access to a file in one directory means an access to another file in that same directory is likely. Secondly, many directory operations (CREATE, LINK) operate on both a parent directory and a child inode at the same time. If the parent and child had nearby SOID numbers, they would likely reside in the same table, simplifying the transaction as discussed in Section 3.5 and Section 3.6.

For these reasons, it would be useful to assign SOIDs such that children of a directory receive SOIDs similar to those of the directory itself. Applied over a whole directory tree, a *namespace flattening policy* would convert "closeness" in the directory hierarchy to "closeness" in SOID values. A number of algorithms could be used for this task; we use a *child-closest policy* [16], which works as follows.

First, the SOID is divided bitwise into a *directory segment* and a *file segment*. The directory segment is further subdivided into a number of *directory slots*. Each slot corresponds to a level in the directory hierarchy, and the value in a slot identifies that directory within its parent. The root directory uses the most significant slot, each of its children the next most significant, and so on. When creating a new directory, the child's directory segment is copied from its parent, with a new value chosen for the most significant empty directory slot.

The file segment is simple sequential counter for files created in that directory. A directory itself has a file segment of all 0s. The first child file of that directory has the same directory segment, but file segment of 1. The second has file segment of 2 and so on. Figure 1 shows an example directory tree and the SOID the child-closest policy assigns to each file or directory in the tree.

This scheme is similar to that used in Farsite, except that the Farsite FileID is variable-length and grows with with directory depth [10]. Supporting variable-length object identifiers would unduly complicate the implementation of Ursa Minor's protocols and components, so we use a fixed-size SOID.

With a fixed-size SOID, the namespace may have both more levels than there are directory slots and more files in a directory than can be represented in the file segment bits. To accommodate this, 2 prefix bits are used to further split the SOID into 4 regions. The first, *primary*, region, uses the assignment policy above. If the hierarchy grows too deep, the too-deep child directory is assigned a new top-level directory slot with a different prefix (the *too-deep* region). Its children grow downwards from there, as before.

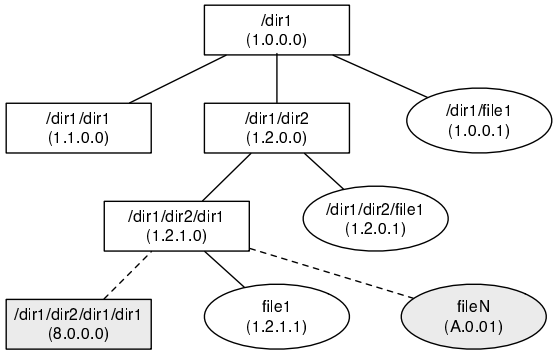

Figure 1: Child-closest SOID assignment policy. The SOID chosen for each element of this simple directory tree is shown. For clarity the example uses a 16 bit SOID and a "." is used to separate the value of each 4 bit directory slot and file segment. The dashed lines show a too-deep directory overflowing to a new "root" and a file in a large directory overflowing to the *too-wide* region.

If there are too many files in a directory and the next directory slot value is unused, the large directory takes over the SOIDs reserved for its nonexistent sibling and the new file is assigned a SOID that would used by its nonexistent cousin. If cousins already exist, the new file is assigned a SOID from the *too-wide* region. Within this region, fewer bits are allocated to the directory segment, and more to the file segment, so more files per directory can be handled. Finally, if either of these additional regions overflow, the *catch-all* prefix is used, and SOIDs are assigned sequentially from this region.

In the case of any overflow, the additional children are effectively created under new "roots" and thus have very different SOIDs from their parents. However, those children will still have locality with their own children (the parent's grandchildren). Thus, one large subtree will be split into two widely separated subtrees, each with locality within itself. If the two subtrees are both large enough, the loss of locality at the boundary between subtrees should not have a significant effect because most operations will be local to one subtree or the other.

By tuning the bit widths of the directory segment, file segment, and directory slots to match the system's workload, instances of overflow can be made extremely rare [16]. Namespace manipulations, such as linking or renaming files, however, will result in the renamed file having a SOID that is not similar to the SOID of its new parent or siblings. The similar situation happens in local disk files systems: a renamed file's inode still resides in its original cylinder group after a rename.

The SOID of a deleted file is available for re-use as soon as the file's storage has been reclaimed from the relevant workers (this step is performed lazily in most cases). Thus, as long as a directory's size does not

change over time, changing its contents does not affect the chance of overflow. In fact, reusing a SOID as soon as possible should provide for a slight efficiency gain, by keeping the metadata B-tree compact.

In all of these cases, outside of the SOID selection policy, MDS treats the SOID as an opaque integer and will operate correctly regardless of how much or little locality the SOIDs preserve. Performance will be be better with higher locality, however. The segment sizes do not need to remain constant over the life of a constellation, or even across the SOID namespace, so there is the potential to adaptively tune them based on the observed workloads, however we have not yet implemented this.

The net effect of combining namespace flattening with SOID-range tables is that each table usually ends up containing a subtree. This is somewhat analogous to the volume abstraction offered by systems like AFS but without the predefined, rigid mapping of subtree to volume. Unlike these systems, a too-large or too-deep subtree will overflow into another table, quite possibly not one served by the same server. One can think of these overflowed subtrees as being split off into separate sub-volumes, as is done in Ontap GX and Ceph.

3.4 Metadata migration

Ursa Minor includes the ability to dynamically migrate objects from one metadata server to another. It does so by reassigning responsibility for a metadata table from one server to another. Because the metadata table (and associated directories) are Ursa Minor data objects accessible to all metadata servers, the contents of the metadata table never need to be copied. The responsibility for serving it is simply transferred to a different server. This section describes the process for doing so in more detail.

Each metadata server exposes an RPC interface via which the delegation coordinator can instruct it to ADD or DROP a table. In order to migrate table T from server A to server B, the coordinator first instructs server A to DROP responsibility for the table. When that is complete, the coordinator updates the delegation map to state that B is responsible and instructs server B to ADD T. At all times, at most one server is responsible.

When server A is instructed to DROP T, it may be in the process of executing operations that use T. Those operations will be allowed to complete. Operations waiting for T will be aborted with an error code of "wrong server", as will any new requests that arrive. Clients that receive such a response will contact the coordinator for a new delegation map. Once the table is idle, server A sets a bit in the table header to indicate the table was cleanly shut down, flushes the table from its in-memory cache, and responds to the coordinator that the table has been dropped.

Adding a table to server B is also simple. When instructed to ADD responsibility, server B first reads the header page of table T. Since T's header page indicates it was shut down cleanly, no recovery or consistency check procedure is necessary, so server B simply adds an entry for T to its in-memory mapping of SOID to table. Any subsequent client requests for SOIDs within T will fault in the appropriate pages of T. Before its first write to T, server B will clear the "clean" bit in the header, so any subsequent crash will cause the recovery procedure to run.

3.5 Multi-object operations

For a server, performing a transaction on a single object is simple: acquire a local lock on the SOID in question and on the SOID's table, perform the operation, and then release all locks.

Performing a transaction with multiple objects or tables within a single server is similar, but complicated by the need to avoid deadlocks between operations that try to acquire the same locks in opposite orders. Each server's local lock manager avoids deadlock by tracking all locks that are desired or in use. When all locks required for an operation are available, the lock manager acquires all of them simultaneously and allows the operation to proceed.

In the more complicated case (shown in Figure 2) of a multi-object and multi-server operation, the server's local lock manager will discover that all the required resources are not local to the server. The lock manager blocks the operation and sets out to acquire responsibility for the required additional tables. To do so, it sends a BORROW request to the Delegation Coordinator. The BORROW request includes the complete list of tables required by the operation; the coordinator's lock manager will serialize conflicting BORROWs. When none of the tables required by a BORROW request are in conflict, the coordinator issues a series of ADD and DROP requests to move all the required tables to the requesting server and returns control to it. Those tables will not be moved again while as the transaction is executing.

When the transaction completes, the requesting server sends a RETURN message to the coordinator, indicating it no longer requires exclusive access to that combination of tables. The coordinator determines whether it can now satisfy any other pending BORROW requests. If so, the coordinator will migrate a RETURNed table directly to the next server that needs it. Otherwise the coordinator can choose to either migrate that table back to its original server (the default action), leave it in place until it is BORROWed in the future, or migrate it to some other server. Note that, while waiting for a BORROW, a server can continue executing other operations on any ta-

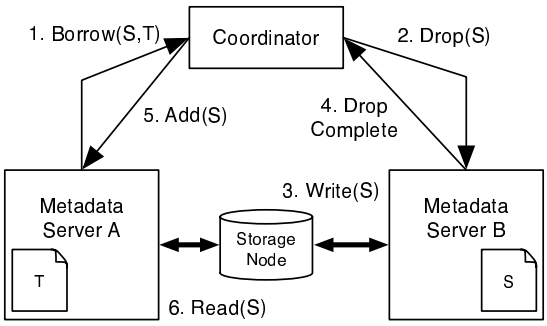

Figure 2: Borrowing a table The sequence of operations required for Server A to handle an operation requiring tables S and T, when table T is initially assigned to server A and table S to server B. Returning to the original state is similar.

bles it already has; only the operation that required the BORROW is delayed.

3.6 Transactions

Underlying the Metadata and Namespace Services is a transactional layer that manages updates to the B-tree structures used for storing inodes and directories. These B-trees are stored as data in Ursa Minor objects. The data storage nodes and their access protocols guarantee that individual B-tree pages are written atomically to the storage nodes and that data accepted by the storage nodes will be stored durably. The transaction system extends these guarantees to transactions involving multiple B-tree pages spread across multiple B-trees.

Atomicity is provided using a simple shadow-paging scheme. All updates to the B-tree data object are deferred until commit time. The data object includes two storage locations for each page, and the location written alternates on every write of that page. Thus, one location will contain the most recent version of that page, and the other location will contain the next most recent version. Each page includes a header that links it to all the other pages written in the same transaction, which will be used by the recovery mechanism to determine whether the transaction committed or needs to be rolled back. Reading a page requires reading both locations and examining both headers to identify the latest version. The server may cache this information, so subsequent rereads only need the location with the latest page contents.

Isolation is guaranteed by allowing only a single transaction to execute and commit on each B-tree at a time. Every transaction must specify, when it begins, the set of B-trees it will operate on. It acquires locks for all of those B-trees from the local lock manager, and holds them until it either commits or aborts. If, during execution, the transaction discovers it needs to operate on a B-tree it does not hold a lock for, it aborts and restarts with the new B-tree added to the set. This strategy is similar to that used by Sinfonia mini-transactions, which share the limitation of specifying their read and write sets up front [4]. Most transactions require only a single execution. The main sources of repeated executions are operations that traverse a file system path: at each step, the SOID of the next directory to read is determined by reading the current directory.

Consistency is only enforced for the key field of the B-tree records; maintaining the consistency of the data fields is the responsibility of the higher level code that modifies them.

Durability is provided by synchronously writing all modified pages to the storage nodes at commit time. The storage nodes may either have battery-backed RAM or themselves synchronously write to their internal disk.

If the metadata server crashes while committing a transaction, it is possible for the B-tree to be in an inconsistent state: for example, only 2 of the 3 pages in the last transaction may have been written to the storage nodes before the crash. To resolve this condition, the metadata server performs a recovery process when it restarts after an unclean shutdown. First, it queries each storage node to determine the location of the last write to the B-tree object (the storage node must maintain this information as part of the PASIS protocol [2]). The location of the last write corresponds to the last page written. Reading that page's header will reveal the identity of all other pages that were part of the same transaction. If all the other pages have transaction numbers that match that of the last written page, then we know that the transaction completed successfully. If any of them has an earlier transaction number, we know that not all page writes were completed, and a rollback phase is performed: any page with the latest transaction number is marked invalid, and its alternate location is marked as the valid one. At the end of rollback, the latest valid version of every page is the same as it was before the start of the rolled-back transaction. The recovery process can proceed in parallel for B-trees with independent updates, whereas two B-trees involved in the same transaction must be recovered together. Because there is at most one transaction committing at a time on a given B-tree, at most one rollback on a given B-tree will be necessary.

3.7 Handling failures

Any of the large number of components of the metadata path can fail at any time, but all failures should be handled quickly and without data loss. In general, our design philosophy considers servers trustworthy; we are primarily concerned with crashes or permanent failure and not with faulty computations or malicious servers.

The most obvious components to consider for failure are the metadata server software and the hardware that

it runs on. A constellation monitoring component polls all metadata servers (as well as other components) periodically, and if the server does not respond within a time-out interval, that metadata server instance is considered to have failed. The monitoring component will then attempt to start a replacement metadata server instance, either on the same hardware or on a different node. The new instance queries the delegation coordinator to determine the tables for which it is responsible and runs the recovery process. After recovery completes, the new instance is in exactly the same state as the previous instance. While the new instance is starting and recovering, client requests sent to the old instance will time-out and be retried.

It is possible that, due to a network partition, a properly operating metadata server may be incorrectly declared by the system monitor to have failed. Restarting a new instance would result in two servers trying to serve the same objects, violating the consistency assumptions. To avoid this, the delegation coordinator revokes the capabilities used by the old instance to access its storage nodes before granting capabilities to the new instance to do the same. Thus, while the old instance may still be running, it will not be able to access its backing store, preserving consistency; nor will clients be able to use capabilities granted by old the instance to access client data. If the revocation attempt fails to reach a quorum of storage nodes, perhaps because they are also on the other side of the network partition, the coordinator will not start a new server instance until the partition heals and the old instance continues uninterrupted until then.

Not only does a failed metadata server affect clients, but it may also affect another server if it failed in the middle of a migration. The delegation coordinator will see its ADD or DROP request time out and propagate this error to any operation that depended on the migration. The metadata being migrated will be unavailable until the metadata server restarts, just like any other metadata served by the failed server. It is reasonable for a multi-server operation to fail because one of the servers it needs is unavailable.

When the failed metadata server restarts, the delegation map it receives from the coordinator will be unchanged from when the server began its last ADD or DROP: a failed ADD will be completed at this time, and failed a DROP effectively never happened. Instead of waiting for a server to restart, the tables assigned to the failed server could simply be reassigned to other working servers. Doing so, however, complicates the process of recovering a table that was involved in a multi-table (but same server) transaction: As described in Section 3.6, both tables must be recovered together, which poses a problem if the two tables have been reassigned to different servers for recovery. Although it is possible to detect and handle this case, in the interest of simplicity, we avoid it by always trying to recover all the tables assigned to a failed server as one unit.

A failed delegation coordinator will prevent the system from performing any more delegation changes, although all metadata servers and clients will continue to operate. As the delegation map is stored in an object and synchronously updated by the coordinator, the coordinator is stateless and can simply be restarted the same as metadata servers. There must be at most one delegation coordinator in a constellation. One method to ensure this, that we have not yet implemented, is to use a quorum protocol to elect a new coordinator [21].

If the failure happened during a migration, the metadata table(s) being migrated will be in one of two states: the delegation map says server A is responsible for table T but server A does not think it is, or the delegation map says no server is responsible for T. The delegation map is always updated in an order such that a server will never be responsible for a metadata table that is not recorded in the delegation map. To handle the first case, a newly started coordinator will contact all metadata servers to determine which tables they are serving and issue the appropriate ADD requests to make the server state match the delegation map. In the second case, an appropriate server is chosen for tables that have no assigned server, and an ADD request is issued.

For storage node failures, we rely on the Ursa Minor data storage protocol to provide fault tolerance by replicating or erasure-coding object data across multiple storage nodes. Since the contents of the metadata tables cannot be reconstructed from any other source, they must be configured with appropriately high fault tolerance.

4 Evaluation

Our goal was to construct a transparently scalable Metadata Service for Ursa Minor. To show we have succeeded, we evaluate the performance of Ursa Minor with a standard benchmark as well as with a range of modified workloads to reveal its sensitivity to workload characteristics. Section 4.1 describes the benchmark's workload, Section 4.2 describes the hardware and software configurations used, Sections 4.3 and 4.4 discuss experimental results, Section 4.5 contrasts these results with the workloads seen in traces of deployed file systems, and Section 4.6 discusses additional observations.

4.1 Benchmark

The SPECsfs97 [35] benchmark is widely used for comparing the performance of NFS servers. It is based on a survey of workloads seen by the typical NFS server and consists of a number of client threads, each of which emits NFS requests for file and directory operations according to an internal access probability model. Each

thread creates its own subdirectory and operates entirely within it. Since each thread accesses a set of files independent from all other threads, and each thread only has a single outstanding operation, this workload is highly parallelizable and contention-free.

In fact, using the namespace flattening policy described in Section 3.3, Ursa Minor is trivially able to assign each thread's files to a distinct SOID range. Thus, each metadata table consists of all the files belonging to a number of client threads, and all multi-object operations will only involve objects in the same table. While this is very good for capturing spatial locality, it means that multi-server operations will never occur for the default SPECsfs97 workload.

Because the SPECsfs97 benchmark directly emits NFS requests, these requests must be translated into the Ursa Minor protocol by an *NFS head-end*. Each head-end is an NFS server and an Ursa Minor client, and it issues a sequence of Ursa Minor metadata and/or data requests in order to satisfy each NFS request it receives. In the default SPECsfs97 workload, 73% of NFS requests will result in one or more Ursa Minor metadata operations, and the remaining 27% are NFS data requests that may also require an Ursa Minor metadata operation. Like any Ursa Minor client, the head-end can cache metadata, so some metadata operations can be served from the head-end's client-side metadata cache, resulting in a lower rate of outgoing Ursa Minor metadata requests than incoming NFS requests. Each head-end is allocated a distinct range of SOIDs for its use, and it exports a single NFS filesystem. Thus, different head-ends will never contend for the same objects, but the client threads connected to a head-end may access distinct objects that happen to be in the same metadata table.

In order to use SPECsfs97 to benchmark Ursa Minor, we found it necessary to make a number of practical modifications to the benchmark parameters and methodology specified by SPEC. First, we modified the configuration file format to allow specifying operation percentages in floating point as necessary for Section 4.4. Second, we doubled the warmup time for each run to 10 minutes to ensure the measured portion of the run did not benefit from startup effects. Neither of these changes should affect the workload presented during the timed portion of the run.

Because we are interested in exploring the scalability of the MDS, we must provision the Ursa Minor constellation so that the MDS is always the bottleneck. Doing so requires enough storage nodes to collectively hold the metadata objects in their caches — otherwise, the storage nodes become the bottleneck. The number of files used by SPECsfs97 is a function of the target throughput and, at high load levels, would require more storage nodes than we have available. Additionally, as the number of files varies, so will the miss-rate of the fixed-size head-end caches, changing the workload seen by the MDS. To avoid these two effects, we use a constant 8 million or 4 million files, requiring 26 GB or 13 GB of metadata. To avoid confusion, we refer to this modified benchmark as *SFS-fixed*.

To maximize MDS load, we configured the NFS head-ends to discard any file data written to them and to substitute zeroes for any file data reads. The Ursa Minor metadata operations associated with the file read and writes are still performed, but the Ursa Minor data operations are not, so we can omit storage nodes for holding file data. In all other regards, including uniform access, we comply with the SPECsfs97 run reporting rules.

4.2 Experimental setup

Table 1(a) lists the hardware used for all experiments, and Table 1(b) lists the assignment of Ursa Minor components to physical machines. This particular assignment was chosen to ensure as uniform hardware and access paths as possible for each instance of a component — every storage node was the same number of network hops away from each metadata server, and each head-end was the same distance from each metadata server.

We configured the test constellations with the goal of ensuring that the MDS was always the bottleneck. The root metadata server was only responsible for objects internal to the MDS (i.e., the metadata for the metadata table objects themselves). The large constellation had 48 NFS head-ends, each serving 20 SFS-fixed client threads (960 in total), and the SOID range assigned to each head-end was split across 8 tables. The resulting 384 tables were assigned in round-robin fashion across metadata servers, such that every head-end used some object on each metadata server. Similarly, tables were stored on 24 storage nodes such that each metadata server used every storage node. These choices increase the likelihood of multi-table operations and contention and are intended to be pessimistic. For small experiments, we used 24 head-ends, 12 storage nodes, and 480 client threads. The SOID assignment policy was configured to support a maximum of 4095 files per directory and 1023 subdirectories per directory, which was sufficient to avoid overflow in all cases. Each storage node used 1.6 GB of battery-backed memory as cache. In addition, 256 MB at each metadata server was used for caching B-tree pages, and the head-ends had 256MB each for their client-side caches.

4.3 Scalability

Figure 3 shows that the Ursa Minor MDS is transparently scalable for the SFS-fixed workload. Specifically, the throughput of the system for both NFS and MDS operations increases linearly as the number of metadata servers increases. This is as expected, because the ba-

Type	Type A	Type B
Count	38	75
RAM	2 GB	1 GB
CPU	3.0 Ghz Xeon	2.8 Ghz Pentium 4
Disk	4× ST3250823AS	1× WD800J
NIC	Intel Pro/1000 MT	Intel Pro/1000 XT
OS	Linux 2.6.26	
Switch	3× HP ProCurve 2848	

(a) Hardware configuration.

Component	Hardware	Large	Small
Storage nodes	Type A	24	12
Metadata servers	Type B	8-32	4-16
NFS head-ends	Type B	48	24
Load generators	Type A	5	2
Root metadata server Root storage node Constellation manager	Type A	1	1

(b) Ursa Minor configuration.

Table 1: Hardware and software configuration used for large and small experiments. The number of metadata servers used varied; all other components remained constant. The root metadata server and its storage node only stored metadata for objects used by the metadata service. Metadata accessible by clients was spread across the remaining storage nodes and metadata servers.

sic SPECsfs97 and SFS-fixed workloads cause no multi-server operations. Thus, adding additional servers evenly divides the total load across servers. Because the head-end servers include caches and because the SFS operation rate includes NFS data requests, the number of requests that reach the metadata servers is lower than that seen by the head-end servers. However, the workload presented to the MDS is much more write-heavy — 26% of requests received by the MDS modify metadata, compared the 7% of NFS requests that definitely will modify metadata and 9% that possibly will.

4.4 Multi-server operations

Since Ursa Minor uses a novel method of implementing multi-server operations, it is important to consider its performance on workloads that are are less trivially parallelizable. To do, so we modified the base SFS-fixed workload to include a specified fraction of cross-directory LINK operations. To keep the total number of operations constant, we reduce the number of CREATE operations by one for every LINK operation we add. Thus, the sum of LINK and CREATE is a constant 1% of the NFS workload. The resulting MDS workload contains a higher fraction of both, because the head-end cache absorbs many of the LOOKUP requests.

We also modify the namespace flattening policy so each client thread's directories are spread across all the tables used by that head-end, giving a $1-(1/N)$ chance any LINK being multi-server. Both LINK and CREATE modify one directory and one inode, so any performance difference between the two can be attributed to the overhead of performing a multi-server operation. A RENAME, however, modifies two directories and their inodes, and is slower than a CREATE even on a single server, which is why we use LINK as the source of cross-directory operations in this experiment.

Figure 4 shows the reduction in MDS throughput for SFS-fixed with multi-server operations compared to SFS-fixed without multi-server operations. Workloads in which 0.05% to 1.00% of NFS operations were cross-directory LINKs resulted 0.07% to 4.75% of MDS operations involving multiple servers. Separate curves are shown for Ursa Minor configurations where each head-end's metadata is split across 16, 8, or 4 tables for a total of 384, 192, or 96 tables in the system. As expected, throughput decreases as the percentage of multi-server ops increases, since each multi-server op requires a table migration. Accordingly, the latency of LINKs are up to 3.5 × that of CREATEs (120 ms vs. 35 ms). Furthermore, when the table is RETURNed to its original server, that server's cache will not contain any of the migrated table's contents. The resulting increase in cache miss rate decreases the throughput of subsequent single-server operations [34].

Additionally, migrating a table makes it unavailable for serving other operations while the migration is in progress. When a single table represents a small fraction of the total metadata in the system, making one table unavailable has a small impact on overall performance. However, if we configure the system to fit the same metadata into fewer tables, the penalty increases, as shown in Figure 4. This is exacerbated in Ursa Minor because threads within the metadata server frequently contend for table-level locks in addition to CPU and I/O. Given that small tables permit finer-grained load balancing, a reasonable Ursa Minor configuration might place 1%-10% of a server's capacity in a single table as suggested for other systems [6]. The major penalty of having far too many tables is that the delegation map will be large, possibly requiring a more efficient means of storing and distributing it.

4.5 Trace analysis

To put the performance of Ursa Minor under multi-server operations into context, we examined two sets of well-studied distributed file system traces to determine what rates of multi-server operations are seen in real-world workloads. Table 2 classifies the operations in each trace

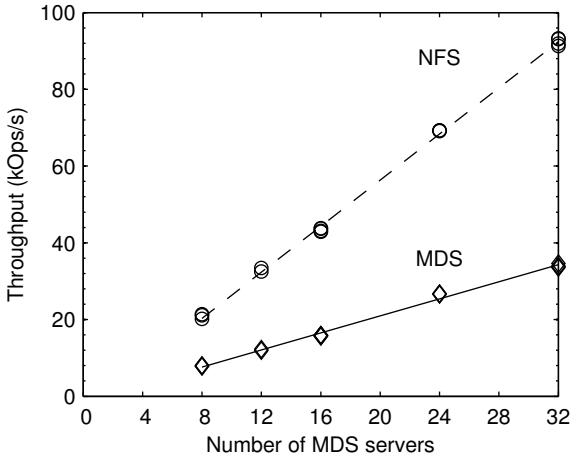

Figure 3: Throughput vs. number of metadata servers. The number of NFS and MDS operations completed per second are shown separately for the SFS-fixed workload. The difference between NFS and MDS operation rates is due to data-only requests and the head-end's metadata cache. The three benchmark runs performed for each configuration are plotted individually. The lines show a linear fit with correlation coefficient of > 0.995. All runs used 8 million files on the large constellation. Our present implementation is limited to about 1 million files per server; plots with fewer files and servers are similar [34].

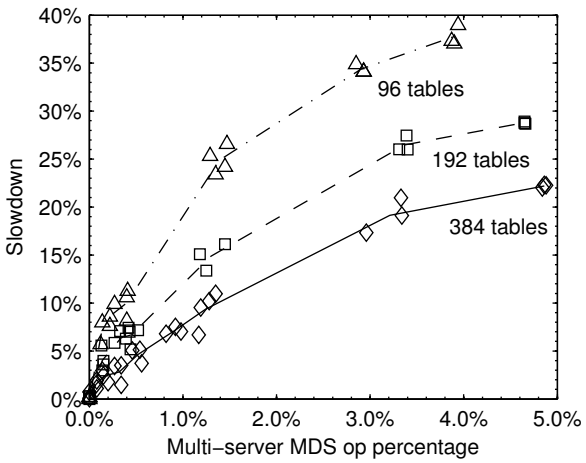

Figure 4: Slowdown vs. percentage of multi-server ops. The slowdown in MDS throughput (compared to a workload with no multi-server ops) is shown for SFS-fixed workloads with varying percentages of multi-server LINKs. The actual percentage achieved in a given run varies from the target percentage; the actual percentage is plotted for each run, and the lines connect the average of all runs with the same target percentage. All runs use 4 million files on the small constellation with 12 metadata servers. The solid line uses the same configuration of 384 total tables used in Figure 3, additional lines use 192 and 96 tables.

by the Ursa Minor metadata operation that would be required to service it. While all operations except for UPDATEs and some LOOKUPs involve more than one object, those objects are almost always in a parent-child relationship. In any system that preserves namespace locality (as the child-closest SOID assignment policy in Ursa Minor does), both objects will be served by the same metadata server. The exceptions are operations on mountpoints, operations on directories that are extremely large, and operations that involve more than one directory. Since the first case should be extremely rare, we expect that cross-directory operations will be the major source of multi-server metadata operations.

The first set of traces are of 3 departmental NFS servers at Harvard University. The workload of each server varied significantly and is described by Ellard et al. [12]. In these traces, RENAME operations may involve more than one directory, and we count cross-directory RENAMEs separately from RENAMEs of a file to a different name in the same directory. Additionally, a LINK operation, while only involving a single file and single directory, might be adding a link in one directory to a file originally created in a different directory. While the original directory does not matter to a traditional NFS server, in Ursa Minor, the file's SOID will be similar to that of its original parent directory, while the new parent directory may have a very different SOID and perhaps be on a different server. Unfortunately, unlike RENAMEs, the LINK RPC does not contain enough information to reliably identify the original parent directory, so we cannot separate these cases. The highest fraction of cross-directory RENAME operations occurred in the DEAS trace and represented only 0.005% of operations in that trace.

The second set of traces are of CIFS traffic to two enterprise-class filers in NetApp's corporate data center. One was used by their engineering department, the other by the their marketing, sales, and finance departments. The fraction of RENAME operations in these traces is similar to those from Harvard, though the distribution of other operations is very different. In the CIFS protocol, the equivalent of the RENAME RPC includes the path of the source and destination directories, so it is possible to determine not only that the the directories are different, but how far apart the source and destination directories are in the directory tree. We were able to analyze a segment of the trace from the Corporate server to calculate rename distances for operations within that segment. Of the 80 cross-directory RENAMEs we found, 56% had a destination directory that was either the immediate parent or child of the source directory.

For comparison, we also show the distribution of operations in the default configuration of SPECsfs97 in Table 2. In all of these workloads, the percentage of cross-directory operations is very low. And, of those cross-directory operations, only a fraction will be multi-server. If directories were assigned randomly to servers, the probability both directories will happen to be on the same server is $1/N$. If the directories involved exhibit spatial locality, as the CIFS traces do, and the OID as-

	EECS	DEAS	CAMPUS	Engineering	Corporate		SPECsfs97
Total operations	180M NFS	770M NFS	672M NFS	352M CIFS	228M CIFS	12.5M CIFS	4.9M NFS
LOOKUP	93.113%	98.621%	97.392%	87.1%	73.2%	62.417%	83.000%
CREATE	0.772%	0.243%	0.286%	0.7%	6.7%	13.160%	1.000%
DELETE	0.810%	0.431%	0.555%	0.006%	0.03%	0.030%	1.000%
UPDATE	5.250%	0.683%	1.766%	1.24%	2.2%	14.282%	14.606%
RENAME (all)	0.054%	0.022%	$< 0.001\%$	0.02%	0.04%	0.036%	0.000%
RENAME (cross-dir)	0.0012%	0.005%	$< 0.001\%$	NA	NA	$< 0.001\%$	0.000%

Table 2: Metadata operation breakdowns for various distributed filesystem traces. The percentage of operations in the original trace that incur each type of Ursa Minor metadata operation is shown. This represents the workload that would seen by the Ursa Minor head-end's metadata cache. Only LOOKUP requests are cacheable, thus we expect the workload seen by the metadata servers to have fewer LOOKUPS. The columns do not sum to 100% because of not all CIFS or NFS operations require Ursa Minor metadata. For the large CIFS traces, the values are calculated from CIFS operation statistics provided by Leung et al. [23] and represent an upper bound for each operation. For the NFS trace and the small CIFS trace, we scan the trace and count the resulting operations. The operations generated by a 5 minute run of SPECsfs97 at 16000 ops/sec are shown for comparison. In all workloads, RENAMES that involve two directories are shown separately and are extremely rare.

signment policy can preserve spatial locality, then both directories are far more likely to be on the same server. Even pessimistically assuming that all cross-directory operations are multi-server, Ursa Minor's approach to multi-server operations can handle an order of magnitude more multi-server operations (.06%) with only a 1.5% decrease in overall throughput compared to a workload with only single-server operations. A system that could execute multi-server operations as fast as single-server ones would be optimal. Even if the workload contains 1% multi-server operations, the slowdown is 7.5%, but such a high rate seems unrealistic, given the rarity of even potentially multi-server operations.

4.6 Additional observations

Our motivation for using migration to handle multi-server operations was that it was the simple solution for the problem at hand. From the starting point of a metadata service that supported migration and single-server operations (over 47000 lines of C code), it only required 820 additional lines of code to support multi-server operations. Of these 820 lines, the global lock manager (necessary for avoiding deadlock) accounted for 530 lines, while the remainder were additional RPC handlers and modifications to the local transaction layer to trigger a BORROW when necessary. In contrast, implementing migration correctly represented 9000 lines of the original metadata server and several months of work.

To provide a basis for comparison, we created a version of Ursa Minor that implements multi-server operations using the traditional 2-phase commit protocol. This version is not nearly as robust or stable as the main version, particularly with regard to handling and recovering from failures, so the 2587 lines required to implement it represent a lower bound. The code to implement a write-ahead log is not included in this total because most other systems include one as part of their basic functionality.

Many of the choices we made in designing the MDS were guided by the properties of the rest of Ursa Minor. Other systems with different underlying storage or failure models might choose to store metadata on the local disks or NVRAM of each metadata server. Migration in such a system would be much more expensive because it requires copying metadata from server to server.

The single delegation coordinator is involved in every multi-server operation, and could become a bottleneck as the constellation scales. We found the coordinator was capable of up to 3500 migrations per second, which is reached with 32 metadata servers and a workload with 1% multi-server ops. Scaling beyond this point would require moving to a hierarchy of coordinators rather than a single one. More details, along with discussion of other workloads and system parameters, are presented in an additional technical report [34].

5 Conclusion

Transparent scalability for metadata is a desirable feature in a large storage system. Unfortunately, it is a difficult feature to provide because it introduces the possibility of multi-server operations, which in turn require relatively complex distributed protocols. By reusing metadata migration to reduce multi-server operations to single-server ones, we were able to implement a transparently scalable metadata service for Ursa Minor with only 820 additional lines of code. Although this approach is more heavyweight than a dedicated cross-server update protocol, the performance penalty is negligible if cross-server operations are are as rare as trace analysis suggests — less than 0.005% of client requests could possibly be cross-server in the traces analyzed. Even if all of those requests were in fact cross-server, Ursa Minor can tolerate an order of magnitude more cross-server operations (.06%) with only a 1.5% decrease in overall throughput. We believe that this approach to handling infrequent cross-server operations is very promising for distributed file systems and, perhaps, for other scalable distributed systems as well.

Acknowledgements

We thank our shepherd, Stephen Hand, and the reviewers for their insightful comments. We thank the members and companies of the PDL Consortium (including APC, EMC, Facebook, Google, Hewlett-Packard, Hitachi, IBM, Intel, LSI, Microsoft, NEC, NetApp, Oracle, Seagate, Symantec, VMware, and Yahoo!) for their interest, insights, feedback, and support. We also thank Intel, IBM, NetApp, Seagate and Sun for hardware donations that enabled this work. This material is based on research sponsored in part by the National Science Foundation, via grant #CNS-0326453, by the Air Force Research Laboratory, under agreement number F49620–01–1–0433, and by CyLab at Carnegie Mellon University under grant DAAD19–02–1–0389 from the Army Research Office. James Hendricks was supported in part by an NDSEG Fellowship, which is sponsored by the Department of Defense. Likun Liu is supported by the Natural Science Foundation of China via project #60963005 and the National Basic Research (973) Program of China via project #2007CB310900.

References

[1] M. Abd-El-Malek, et al. Ursa Minor: versatile cluster-based storage. Conference on File and Storage Technologies. USENIX Association, 2005.

[2] M. Abd-El-Malek, et al. Fault-scalable Byzantine fault-tolerant services. ACM Symposium on Operating System Principles. ACM, 2005.

[3] A. Adya, et al. FARSITE: federated, available, and reliable storage for an incompletely trusted environment. Symposium on Operating Systems Design and Implementation. USENIX Association, 2002.

[4] M. K. Aguilera, et al. Sinfonia: a new paradigm for building scalable distributed systems. ACM Symposium on Operating System Principles. ACM, 2007.

[5] D. C. Anderson, et al. Interposed request routing for scalable network storage. Symposium on Operating Systems Design and Implementation, 2000.

[6] T. E. Anderson, et al. Serverless network file systems. ACM Symposium on Operating System Principles. Published as *Operating Systems Review*, **29**(5):109–126, 1995.

[7] S. Baker and J. H. Hartman. *The Mirage NFS router*. Technical Report TR02–04. Department of Computer Science, The University of Arizona, November 2002.

[8] B. Callaghan, et al. *RFC 1813 - NFS version 3 protocol specification*, RFC–1813. Network Working Group, June 1995.

[9] A. L. Chervenak, et al. Protecting file systems: a survey of backup techniques. Joint NASA and IEEE Mass Storage Conference, 1998.

[10] J. R. Douceur and J. Howell. Distributed directory service in the Farsite file system. Symposium on Operating Systems Design and Implementation. USENIX Association, 2006.

[11] M. Eisler, et al. Data ONTAP GX: a scalable storage cluster. Conference on File and Storage Technologies, 2007.

[12] D. Ellard, et al. Passive NFS tracing of email and research workloads. Conference on File and Storage Technologies. USENIX Association, 2003.

[13] S. Ghemawat, et al. The Google file system. ACM Symposium on Operating System Principles. ACM, 2003.

[14] G. A. Gibson, et al. A cost-effective, high-bandwidth storage architecture. Architectural Support for Programming Languages and Operating Systems. Published as *SIGPLAN Notices*, **33**(11):92–103, November 1998.

[15] J. N. Gray. Notes on data base operating systems. In , volume 60, pages 393–481. Springer-Verlag, Berlin, 1978.

[16] J. Hendricks, et al. *Improving small file performance in object-based storage*. Technical report CMU-PDL-06-104. Parallel Data Laboratory, Carnegie Mellon University, Pittsburgh, PA, May 2006.

[17] D. Hitz, et al. File system design for an NFS file server appliance. Winter USENIX Technical Conference. USENIX Association, 1994.

[18] J. H. Howard, et al. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems (TOCS)*, **6**(1):51–81. ACM, February 1988.

[19] W. Katsurashima, et al. NAS switch: a novel CIFS server virtualization. IEEE Symposium on Mass Storage Systems. IEEE, 7–10 April 2003.

[20] A. J. Klosterman and G. R. Ganger. *Cuckoo: layered clustering for NFS*. Technical Report CMU–CS–02–183. Carnegie Mellon University, October 2002.

[21] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, **16**(2):133–169. ACM Press, May 1998.

[22] E. K. Lee and C. A. Thekkath. Petal: distributed virtual disks. Architectural Support for Programming Languages and Operating Systems. Published as *SIGPLAN Notices*, **31**(9):84–92, 1996.

[23] A. W. Leung, et al. Measurement and analysis of large-scale network file system workloads. USENIX Annual Technical Conference. USENIX Association, 2008.

[24] Lustre, Apr 2006. http://www.lustre.org/.

[25] M. K. McKusick. Running 'fsck' in the background. BSDCon Conference, 2002.

[26] M. N. Nelson, et al. Caching in the sprite network file system. *Transactions on Computer Systems*, **6**(1):134–154. ACM, February 1988.

[27] J. Norton, et al. *Common Internet File System (CIFS) Technical Reference*. SNIA, 12–12 March 2002.

[28] When to Use Transactional NTFS, Apr 2006. http://msdn.microsoft.com/library/en−us/fileio/fs/when_to_use_transactional_ntfs.asp.

[29] S. V. Patil, et al. GIGA+: Scalable Directories for Shared File Systems. ACM Symposium on Principles of Distributed Computing. ACM, 2007.

[30] H. Patterson, et al. SnapMirror: file system based asynchronous mirroring for disaster recovery. Conference on File and Storage Technologies. USENIX Association, 2002.

[31] Reiser4 Transaction Design Document, Apr 2006. http://www.namesys.com/txn-doc.html/.

[32] F. Schmuck and R. Haskin. GPFS: a shared-disk file system for large computing clusters. Conference on File and Storage Technologies. USENIX Association, 2002.

[33] S. Shepler, et al. *Network file system (NFS) version 4 protocol*, RFC–3530. Network Working Group, April 2003.

[34] S. Sinnamohideen, et al. *A Transparently-Scalable Metadata Service for the Ursa Minor Storage System*. Technical report CMU-PDL-10-102. Parallel Data Laboratory, Carnegie Mellon University, March 2010.

[35] SPEC SFS97 R1 V3.0 Documentation, Jan 2010. http://www.spec.org/sfs97r1/.

[36] C. A. Thekkath, et al. Frangipani: a scalable distributed file system. ACM Symposium on Operating System Principles. Published as *Operating Systems Review*, **31**(5):224–237. ACM, 1997.

[37] S. A. Weil, et al. Ceph: A scalable, high-performance distributed file system. Symposium on Operating Systems Design and Implementation. USENIX Association, 2006.

[38] B. Welch, et al. Scalable performance of the Panasas file system. Conference on File and Storage Technologies. USENIX Association, 2008.

[39] K. G. Yocum, et al. Anypoint: extensible transport switching on the edge. USENIX Symposium on Internet Technologies and Systems. USENIX Association, 2003.