

File Creation Strategies in a Distributed Metadata File System

To appear in the Proceedings of IPDPS '07, Long Beach, CA, March 2007

Ananth Devulapalli
Ohio Supercomputer Center
1 South Limestone St., Suite 310
Springfield, OH 45502
ananth@osc.edu

Pete Wyckoff
Ohio Supercomputer Center
1224 Kinnear Road
Columbus, OH 43212
pw@osc.edu

Abstract

As computing breaches petascale limits both in processor performance and storage capacity, the only way that current and future gains in performance can be achieved is by increasing the parallelism of the system. Gains in storage performance remain low due to the use of traditional distributed file systems such as NFS, where although multiple clients can access files at the same time, only one node can serve files to the clients. New file systems that distribute load across multiple data servers are being developed; however, most implementations concentrate all the metadata load at a single server still. Distributing metadata load is important to accommodate growing numbers of more powerful clients.

Scaling metadata performance is more complex than scaling raw I/O performance, and with distributed metadata the complexity increases further. In this paper we present strategies for file creation in distributed metadata file systems. Using the PVFS distributed file system as our testbed, we present designs that are able to reduce the message complexity of the create operation and increase performance. Compared to the basecase create protocol implemented in PVFS, our design delivers near constant operation latency as the system scales, does not degenerate under high contention situations, and increases throughput linearly as the number of metadata servers increase. The design schemes are applicable to any distributed file system implementation.

1 Introduction

As processors and interconnects continue to deliver exponentially better performance as time goes on, storage has been relatively stagnant, and I/O is becoming a painful bottleneck in high-performance computing. Due to the fundamental limitations of rotating magnetic media, one of the

few possible ways to increase throughput of data operations is by employing multiple devices in parallel. Striping data across many disks or data servers is a straightforward way of scaling bandwidth; however, applications do not only read and write *data*, they also manipulate *metadata* to organize their data in a typical tree hierarchy. In many fields, accessing and modifying the metadata consumes a significant fraction of the overall run-time [16]. Furthermore, scaling metadata performance by aggregating servers and storage devices is inherently more complex than data striping.

In this paper, we analyze strategies for file creation in file systems that distribute the metadata across multiple servers. There are many steps involved in a single create operation because the data objects, metadata representation, and parent directory entry may all reside on separate machines. Of the set of typical metadata operations, *create* features a high level of complexity due to the number of disparate objects involved and the constraints imposed by correctness guarantees. While designing for good create performance, it is important to ensure that no other file system client can see the system in an inconsistent state.

The rest of this paper starts with an overview of distributed file system architecture in Section 2. The design choices for object creation are described in Section 4, followed by experimental results in Section 5. Section 8 summarizes the contributions of this work and suggests directions for future research.

2 Background

Previous studies have shown that high-end computing workloads feature data accesses by multiple processes as well as high metadata rates, and frequent and concurrent creates and deletes [20]. Metadata operations make up over half of some workloads [16]. As ever larger machines with more disks are being deployed [23], a single metadata server is no longer sufficient to handle the workload. Some of the metadata issues can be ameliorated by using collective interfaces, such as MPI-I/O [21], at the clients [9]. These techniques limit accesses to one client, with results

broadcast outside of the file system to the other cooperating clients. Unfortunately, not all usage scenarios can limit their metadata accesses in this way, and non-collective I/O interfaces, such as POSIX [6], are still prevalent.

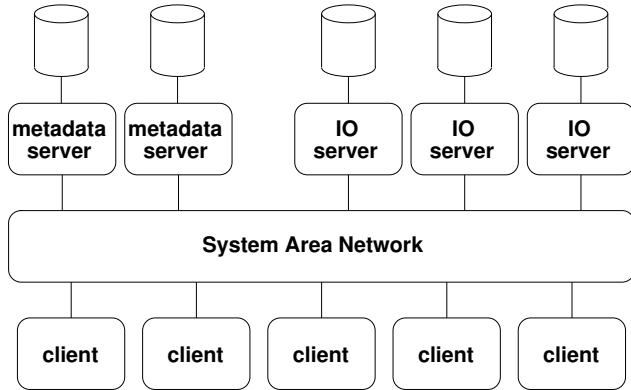


Figure 1. Distributed file system architecture.

A common distributed file system architecture consists of many *IO servers* that store data contents, one or more *metadata servers* that store information about the data, and many clients, all of which are connected by a shared network as shown in Figure 1. Clients typically communicate with both IO and metadata servers to perform file system activities, and servers may or may not communicate among themselves. This architecture is used in many current parallel file system implementations [8, 3, 12].

Some systems delegate operations, such as create, from the requesting client to a single server, that in turn contacts other servers as required to perform the metadata and data operations [3]. This approach simplifies the client implementation at the expense of scalability. It also requires an architecture where all servers can communicate with each other. In this paper, we consider the more general case where clients communicate directly with the servers involved in a transaction.

Our testbed for implementation is PVFS [2, 8] (version 2), an open source parallel and distributed file system. PVFS uses striping across IO servers to achieve high data rates, and fully distributes file, directory, symbolic link, and other metadata objects across one or more metadata servers. Operation is optimized for the case of cooperating clients, avoiding the need for mandatory defensive locking, but also not allowing for the use of client-side data caches. The server is stateless from the protocol point of view, although network connections are cached for performance. Distributing metadata among multiple servers ensures good scalability [9] but at a cost of an increased number of transactions required to perform a single operation from the client point of view. Furthermore, PVFS is designed with correctness in mind, so each of the multiple transactions is followed by

a disk synchronization to ensure data stability, and relationships between the transactions forces serialization at times.

The contribution of this paper is an analysis of the requirements of the *file create* operation in the distributed file system architecture of Figure 1. We present multiple designs to decrease individual create operation latency and increase overall create throughput for multiple clients, while ensuring that the system remains consistent.

3 Basic Create Protocol

We begin with a description of the components of a file. A single file striped across the IO servers exists as multiple entries (disk files and database rows) in the IO and metadata servers. First, there are generally N *datafiles*, one on each IO server, that hold chunks of the data in the file. We call these *datafiles*, and the 64-bit descriptors used to refer to them are *IO handles*. Next there is a *metafile* on a metadata server that contains a list of all the *IO handles* that comprise the file. This *metafile* has a *metahandle* that represents it, and an associated set of attributes for the file, such as ownership credentials and permission bits. The *metahandle* of the file is added to the list of *metahandles* maintained by the parent directory, making the file visible. The parent directory need not exist on the same metadata server as the file itself. Table 1 summarizes the components of a file.

Number	Description
N	File data chunks
1	Metafile, with attributes
1	Directory entry

Table 1. Distributed file components.

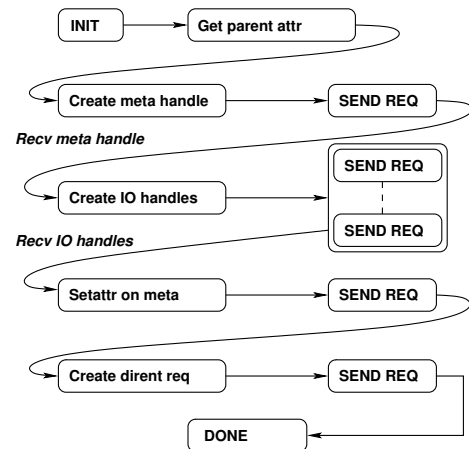


Figure 2. Create state machine.

Figure 2 illustrates the sequence of operations performed

by a PVFS client to complete a file create operation, after it gets the attributes of the parent directory:

1. Select a metadata server at random, create a *metafile* at that server and receive corresponding *metahandle*.
2. Select a set of IO servers, create a *datafile* at each server and receive corresponding *IO handles*.
3. Set the attributes of the *metafile*, including the list of *IO handles* of the created *datafiles*.
4. At the metadata server hosting the parent directory, create a directory entry for the newly created file.

As seen in the figure, there are four serial steps for the create operation, each of which involves one or more request and response pairs to IO or metadata servers. We define a step as an entity comprised of an operation and its subsequent communication state. For example, the “create meta handle” step refers to the “create meta handle” state along with the consecutive “SEND REQ” state. The state after “create IO handles” depicts multiple “SEND REQ” states, which implies there might be many IO servers involved requiring multiple messages

Parallelism does occur at the *IO handle* create step in Figure 2, where each IO server operates independently. Although the figure does not show it explicitly, each of these operations is followed by a test for success, and in the failure case, all preceding operations are carefully undone by the invoking client. The states are ordered so that the directory entry is not created until after the file itself is created so that other clients can not view an inconsistent state. Due to the multiple serial steps in the basic create protocol, this operation becomes relatively expensive when compared to single-server file systems such as NFS [13]. Good metadata scalability comes at the expense of degraded performance for a single operation.

4 Design Alternatives

This section describes techniques to achieve improved metadata performance while retaining good scalability, approached from the view of incrementally improving the create operation.

4.1 Compound operations

Looking at the list of fundamental steps in the basic create protocol, it can be seen that the initial creation of the *metahandle* is independent from the creation of the *IO handles*, thus these two operations can proceed in either order, or concurrently. If run in parallel, there will be some savings from avoiding a serialization, but swapping the order so that *IO handles* are created first moves the *metahandle*

create and *metahandle* setattr operations beside each other, which can then be combined into a single operation. Using a compound operation like this also avoids one serialization step, and has the added advantage of saving an *RPC*.

Compound operations have been used frequently in the past. As selective combinations for enhanced functionality, they can be used to avoid race conditions in distributed applications, such as in the *test and set* or *fetch and add* operations in shared-memory machines [10] and with the *RDMA write and invalidate* network [14]. Other uses of compound operations are purely to reduce transaction cost, as in NFSv4 [19] where most operations can be glued together into a compound and issued as a single *RPC*. Extensions to POSIX [5] introduce a combined *getattr* plus *stat* to address one particular performance problem that occurs in the frequently used directory listing operation.

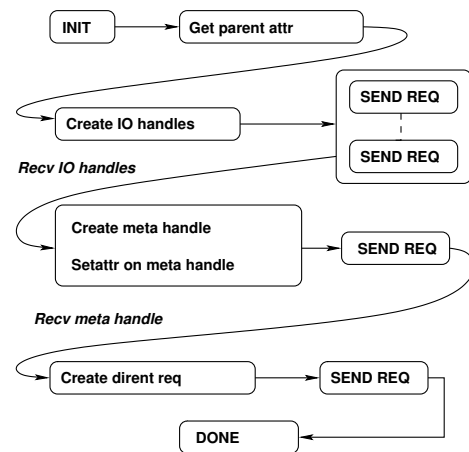


Figure 3. Create state machine with compound operations.

Here, we combine the *create* and *setattr* operations as a single new server command that can be used in multiple scenarios, but we do not introduce a generic facility for operation compounding as there are not enough situations that can exploit it to justify the implementation cost currently. The modified state machine that saves one *RPC* call is shown in Figure 3. As will be seen in Section 5, the gain from this modification is modest, and considerations of complexity may rule against widespread use of compound operations except in specific cases.

4.2 Handle selection strategies

Microbenchmarks and timing analysis suggest that the creation of *IO handles* is a serious contributor to overall create time. The process involves communicating with a potentially large number of IO servers, depending on the striping mechanism used by the file system. While all the *IO han-*

dle creates can proceed independently, they must all finish before creation of the *metahandle* starts, because the *metahandle* lists the newly-created *IO handles* returned from the IO servers.

The only way to parallelize the *IO handle* and *metahandle* creation steps is if the client knows the *IO handles* in advance. Some approaches to solve this problem are discussed below, after some information on possible handle schemes is presented.

Handle mapping Handles are opaque bit sequences used to refer to a particular object, usually 64 or 128 bits in length. In PVFS, handle ranges are statically assigned at file system creation. Each client loads a configuration file upon first access that maps linear handle ranges to particular metadata or IO servers. (The handle space for *metafiles* and *datafiles* are distinct.) Different handle assignment algorithms are implemented by other file systems. PNFS [4] uses a central directory to map handles into storage servers—a client must contact the main server to find out the data server that holds its object. The experimental system Ceph [24] uses a well-known hashing function so that any client can perform the algorithm used to convert handles to storage servers. To achieve good scalability, it is necessary that clients be able to select their own handles for newly created objects, yet avoid any centralized handle allocation point.

Handle guessing For fixed handle ownership ranges such as PVFS uses, a client can guess a potential *IO handle* and suggest to the IO server that it allocate the suggested handle. This approach would work for reversible hash-based policies too. The major risk to this approach, though, is that of handle collision. Multiple clients may be guessing new handles at the same time. This scheme begs for development of a “back-off” protocol that can jump ahead in the available handle space hoping to find a free one. It also adds considerable complication to the state machines at both the client and at IO servers for processing new object allocation.

Central directory A file system may choose to implement a centralized directory of allocated and free handles. For tightly coupled systems, this may be the most efficient way for clients to request new handles. The scheme has the downside that both clients and servers must access a single list to perform handle allocation, although this may not be problematic for tightly coupled systems.

Handle reservation Clients may request, in advance, new handles from the IO servers, before they have a need to create new objects. By reserving a set of handles in advance, individual create operations can proceed much more

quickly. However, this approach leads to many more questions. When should clients reserve handles, and how many should they request? What happens to reserved yet unused handles when clients exit? Should the IO servers create and commit to disk a *datafile* for each reserved handle? We pursue this approach in the following subsections and discuss the issues that arise.

4.3 Leased handles

We first present and analyze a scheme whereby clients can lease handles in advance of create time, and later request that one of those handles be assigned to newly created *datafiles*. The scheme is illustrated in Figure 4. As shown, the state machine has two paths, a *fast path* that completes the entire create operation in two steps and a *slow path* that takes three steps.

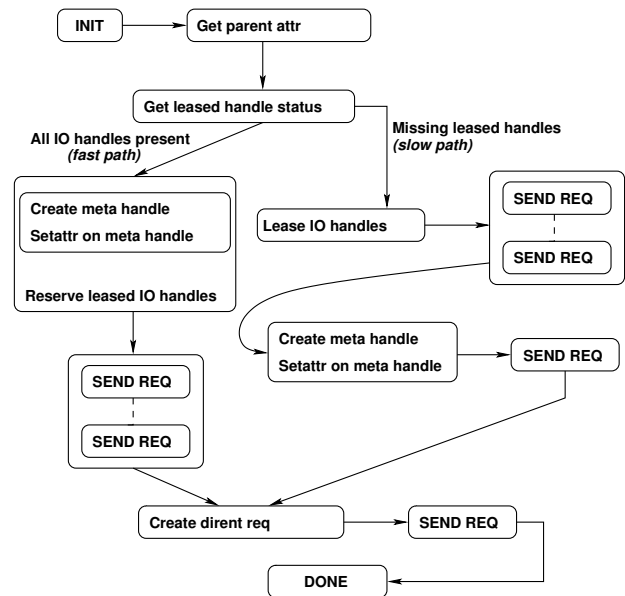


Figure 4. Create state machine with leased handles.

Under this scheme, the initial steps up through getting the attributes of the parent are the same as in the basic create protocol. Next, for each of the selected IO servers, the client checks if it possesses any *leased handles* that can be used. If the client finds that it has a leased handle for all IO servers involved in the operation, it takes the *fast path* where it can concurrently execute the following steps:

1. Request the IO servers to reserve the leased handle for a create. If the client determines that any IO server is running low on leased handles, it can further request the server to send another set of leased handles embedded in the response.

2. Use the leased handles for creating *metahandle* and setting attributes on the *metahandle*.

The actions in step 1 have a further positive side-effect: since a client can initiate requests for an additional set of leased handles for servers whose leased handle count falls below a threshold level, at no time in future, bar failures or exceptions, will a client need to travel down the *slow path*. That is, once a client has used all IO servers in the system as targets for create operations just once, it will always take the *fast path*.

The *reserve leased handle* request is similar to the original *create handle* request, in that a server allocates a new *datafile* on disk, but uses the handle identifier given by the client instead of creating a new one.

If there exists at least one server for which the client has no leased handles, the *slow path* is taken. On this path, a request is sent to each IO server for which the client does not have leased handles. (A reserve request is sent to the other servers as in the fast path.) Upon receiving a lease request, an IO server allocates a small number of handles with the first handle among those allocated reserved for the create operation in progress. Note that request for leased handles are serialized at the server which guarantees atomicity.

The client exits both the slow and fast paths with a set of leased handles for future create operations on the participating IO servers. It also has created the *metafile* and set its attributes including the list of *IO handles*. Finally, a *create dirent* request is sent to the metadata server hosting the parent directory.

4.3.1 Leased handle correctness

In a file system, a file is visible in the system only when all operations succeed. If any one of the intermediary steps fails, the entire create operation must fail. Since a file can only be reached by another client when a directory entry exists for it, the leased handle scheme can be seen not to violate correctness as the *create dirent* request is the last step and only happens if all the previous steps succeed.

Under the leased handles scheme, only the *IO handles* of a file are pre-fetched. If a client with leased handles dies, those *IO handles* are lost, but no file is created. Similarly if a client disappears after creating the *metahandle*, the only side-effect will be orphan handles, which can be reclaimed later but are not seen by other file system clients. If a client tries to reserve a leased handle on failed IO server, that operation will fail, resulting in failure of the create operation. Similarly failure to either create a *metahandle* or adding the directory entry will fail the entire operation, ensuring consistency.

4.3.2 Leased handle leaks

Though the algorithm is correct, particular failures could result in lost *IO handles*, orphaned *metahandle*, or both. These problems can be easily solved by the file system check (`fsck`) utility, which can discriminate between associated *metahandle* and *IO handles*, and orphan *metahandle* and leaked *IO handles*. The problem with `fsck` is that it is a burden to require its operation to clean up such leaks, and can interfere with normal functioning of the system.

While the original file system implementation also has many opportunities to create orphaned *metafiles* and *datafiles*, our leased handles algorithm increases the potential number of leaks. In a system with N clients, with up to L leased handles each at any given moment, in the worst case, the number of leased handles is $N \times L$, which can be quite large for big machines. We did not discuss above a wise protocol extension to return leased handles to the server before a client exits, but even with that functionality, failed clients will cause leaks.

One solution to this problem is to associate a timeout with each leased handles. Whenever an IO server allocates a set of leased handles, it selects a time range for which the handles can be used and returns this value to the leasing client. A leased handle must be used within the timeout period. When a handle times out, the server invalidates the lease and marks the handle free. Clients also track the timeout for each set of leased handles and treat expired handles as nonexistent, potentially falling back to the slow path in Figure 4 if its handles have expired.

4.4 Pre-creation of datafiles at IO servers

The leased handles scheme of the previous section is able to overlap creation of the *metahandle* and *IO handles*, reducing the number of steps to two in the fast path. However, the number of request and response messages remains the same. One way to reduce the number of messages is to go beyond leasing *IO handles*, and actually pre-create the *datafiles*. This scheme is naturally designed on top of the leased handle scheme. On reception of a leased handle request from the client, an IO server can also create *datafiles* for those handles, and respond to the client with leased handles only after all the *datafiles* corresponding to the leased handles are committed to disk. This guarantees to a client with leased handles that the respective *datafiles* are ready for use, allowing it to skip communication with IO servers altogether during future create operations.

Figure 5 shows the modified state machine with *datafile* pre-creation. Like the leased handles scheme, there are two paths. The *slow path* is identical as before. The fast path omits the need to create *datafiles* corresponding to particular leased handles, and only creates the *metafile* on the metadata server. After either path, the directory entry is

created on the parent directory metadata server as in all the schemes. One more distinction from the leased handles scheme is that the client must fall back periodically to the *slow path* to request new leased handles with pre-created *datafiles*, since there is no regular communication at each create step as in the leased handles scheme.

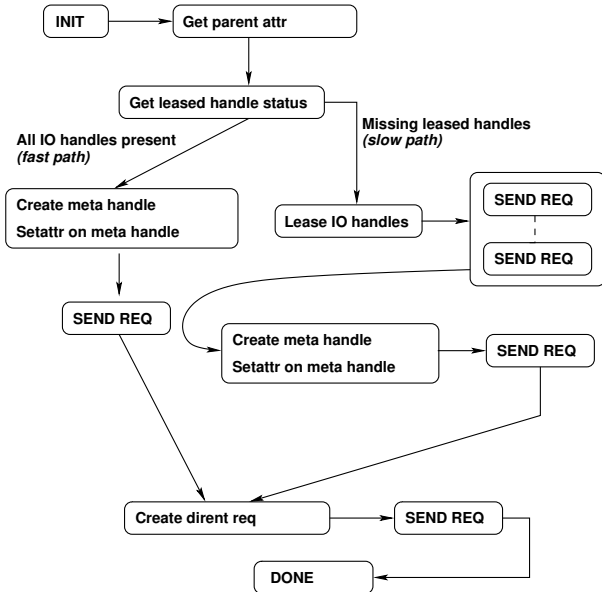


Figure 5. Create state machine with leased handle pre-creation.

Though the average number of steps is more than two in this scheme, the overall complexity of the create operation is reduced. Given a create operation involving N IO servers, the number of request and response message pairs for each create in the leased handles scheme is on average $N + 2$, for the reservation requests to the IO servers and one each for creation of the *metafile* and creation of the directory entry. In the *datafile* pre-creation scheme with each client requesting L pre-created *datafiles*, there are $L - 1$ trips down the fast path requiring two messages each, and one trip down the slow path requiring $N + 2$ messages, for an average of $2 + \frac{N}{L}$. For clients requiring large create rates, L can be tuned to be large, bringing the average closer to two messages per create.

4.4.1 Pre-creation correctness

The argument for correctness of *datafile* pre-creation follows the same logic as leased handles. Since failure of any intermediate step will fail the overall create operation, a file will be visible only when all the steps succeed. But, paradoxically, successful completion of the create operation can also occur even if an IO server has died, as there is no need

to contact that particular IO server during create operation. Still, the file system is consistent since the state that occurs after such a “dead IO server” create is no different than the state that would occur if the IO server dies *after* the file create. Of course, trying to access the file would fail in both cases, but when the IO server recovers, the file system will be consistent without the need for an `fsck`, because the pre-created *datafile* was committed to disk just like a normal *datafile* in the basecase scheme.

4.4.2 Pre-creation leaks

Once an IO server pre-creates *datafiles* and returns the corresponding IO handles to a client, there is no communication between it and the client until the client tries to access or modify one of those *datafiles*, or the client requests for a new set of leased handles with pre-created *datafiles*. Unlike in the leased handles scheme, here, an IO server never gets a positive acknowledgment that one of its pre-created *datafiles* is part of an existing file rather than just pending creation. Because of this, a simple timeout scheme can not be used to recycle pre-created *datafiles*. However, we can limit the extent of possible leaks. First, a client can piggy-back a *creation confirmation* message to the IO server when it goes to write data to the newly created file. A simple bit in the IO request would suffice for this. (In the case of cooperating parallel clients, any or all clients would set that bit on writes of collectively created files.) Second, clients can return unused leased handles causing deallocation of the corresponding pre-created *datafiles*, as mentioned in the leased handles scheme.

However, the number of unused but pre-created *datafiles* can still grow without bound. Consider a client that creates many files using its pre-created *datafiles* but never acknowledges this to the IO server by writing to the files, and then crashes. The only way for the IO server to find out the true disposition of the files is through an `fsck`-like operation where the *metafiles* indicate which *datafiles* are in use. We can limit the need for full file system checks by keeping a timeout for each pre-created *datafile* on the IO server, and mandating that clients must acknowledge the use of each pre-created *datafile* before the timeout. This explicit acknowledgment case would be rare, and needed only in the case of allocation of many empty files. Now if the IO server finds that the number of potentially leaked pre-created *datafiles* is large, it can send a request to all the metadata servers asking to validate which of the given data handles are actually in use.

4.5 Discussion

The three designs of the file system create operation improve upon the basecase by reducing round-trip messages and increasing parallelism, while preserving correctness of

the overall file system. The “compound” scheme is straightforward to implement but offers only minimal overall operational savings. Both the leased handles and *datafile* pre-creation schemes offer more substantial potential performance improvements yet introduce the complexities of managing handle or *datafile* leaks. The *datafile* pre-creation scheme is unique because, on average, it scales at $O(1)$ rather than $O(N)$ with the number of IO servers.

For the schemes that require pre-creation of handles or *datafiles*, the number of leases per client should be chosen well to provide good performance yet minimize startup overheads and expenses of cleaning up leaked objects. Rather than using a fixed number of handles, each client can monitor its own behavior in terms of create rate and request a number of handles that increases with apparent load. There is also a potential need to monitor IO server load, as *datafile* pre-creation can serve to limit IO server load by effectively batching requests, regardless of whether clients are experiencing high create rates.

5 Experiments and results

Experiments were run on 30 identical nodes, each of which has dual Opteron 250 processors with 2 GB of RAM and an 80 GB local SATA disk, running Linux version 2.6.17.6. Each node has a Tigon3 Gigabit Ethernet NIC and are connected by a single non-blocking Gigabit Ethernet switch. The round-trip small packet latency between two nodes is approximately 80 μ s, and the sustained unidirectional throughput is about 950 Mb/s.

For our experiments four versions of the create protocol were evaluated.

Basecase: Basic create protocol, Section 3.

Compound: Create protocol with compound operations, Section 4.1.

Leased: Leased handles algorithm, Section 4.3

Pre-creation: Pre-creation of *datafiles*, Section 4.4

The number of leased handles in Leased and Pre-creation protocols was fixed at 20 for all our experiments.

As discussed earlier in Section 2, we use PVFS version 2 as our experimental testbed. For our experiments, the create protocol implemented in PVFS2 version 1.5.1 served as the basecase and the rest of the protocols were derived from it. Drawing conclusions from experiments that involve physical disks are difficult due to variations in disk access times. Also, to store its metadata, PVFS uses the Berkeley DB4 database, which can cause unpredictable accesses based on the state of its internal data structures. Therefore we not only report the performance when metadata is on-disk but also when metadata is written to a RAM-based file system,

called `tmpfs`. We discuss the disk variability problem further in Section 7.

The goal of the experiments is to evaluate the scalability and the performance of the algorithms under both low and high client usage scenarios. The three variable parameters are the number of IO servers, metadata servers and clients. The experiments and results are discussed in the rest of this section.

5.1 Single-operation latency

The purpose of this experiment was to test the differences in response time of the four protocols as the number of IO servers is varied. For this experiment we used a single metadata server, one client and changed the number of IO servers from one to 26. The core of the client test program creates a single file striped across all available IO servers, recording the overall time to perform that operation, then deletes it, looping 1000 iterations to collect statistical information and with an un-timed warm-up of 20 iterations to ensure that all network connections are up.

Figure 6 shows the plot of response times of four protocols on `tmpfs` and on disk. The standard deviations on disk are so large that all algorithms appear statistically identical, whereas on `tmpfs` we see clear differences among the schemes. On `tmpfs`, the response times of basecase, compound and leased handles increase monotonically as the number of IO servers is increased; however, the response time of pre-creation is roughly constant.

The behavior of the algorithms is as expected from the design. Compound saves one *RPC* over basecase, and we see roughly 100 μ s difference between the two, in the neighborhood of a round-trip time on our network. Leased handles scheme overlaps *metafile* creation with the creation of *datafiles*. As long as *metafile* creation time dominates, we see a one-round-trip difference between those two curves, but as the number of IO servers increases, *datafile* creation begins to dominate and the gap between the two curves reduces for high numbers of IO servers. This is possibly due to the increased cost of message response handling at the single client as the number of IO servers increases. In the case of pre-creation, the average response time increases at a much slower pace than the rest, since in steady state pre-creation involves only two messages. But the standard deviation keeps increasing, since the maximum response time increases as the number of IO servers is increased.

5.2 Multiple-client throughput

This experiment was a stress test for all algorithms. For this experiment we fixed the number of IO servers at four, used one metadata server, and varied the number of clients. All the clients create unique files under a single directory,

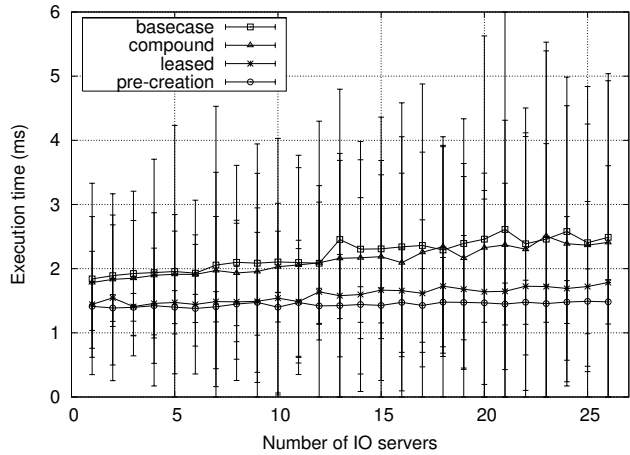
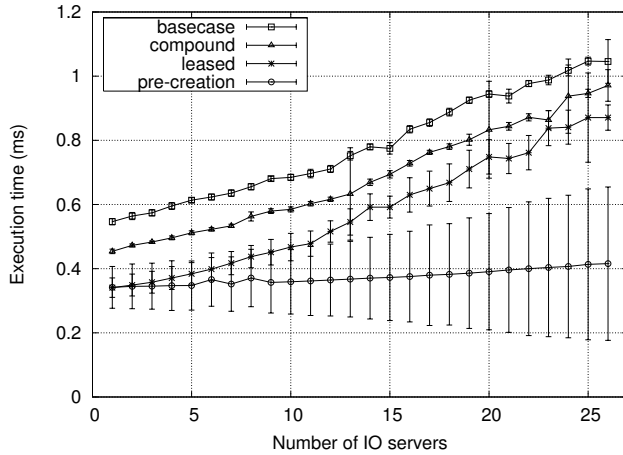


Figure 6. Mean latency on (a) tmpfs, (b) disk; different y-axis scales.

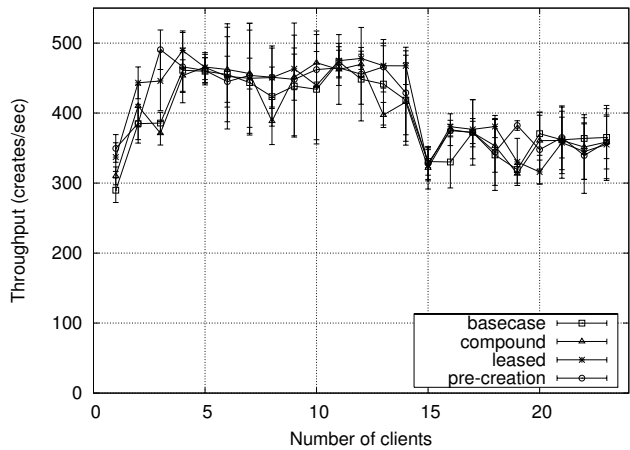
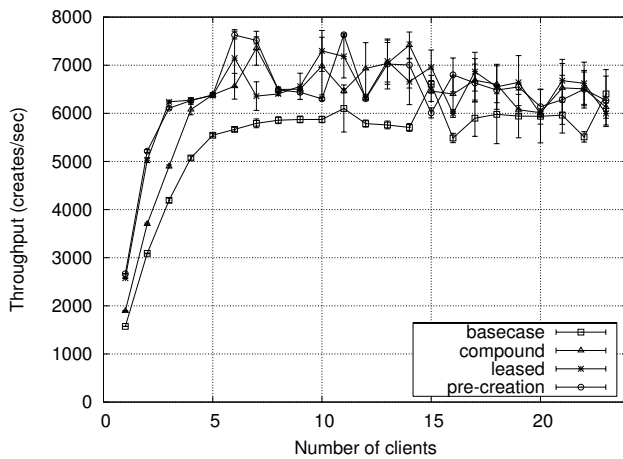


Figure 7. Create operation throughput on (a) tmpfs, (b) disk; different y-axis scales.

which means all creates are serialized at the single metadata server. The core of the experiment involves all clients arriving at a barrier, the leader among the clients starting the timer, each client creating about 120 files, the clients reaching another barrier, the leader stopping the timer and finally each client deleting the created files. This kernel was repeated 100 times while collecting average and standard deviations. We calculate the number of files created in a given amount of time by multiplying the number of files per client by the number of clients and dividing by the elapsed time for execution, including the cost of the barriers.

Figure 7 shows the throughput on tmpfs and disk. We see again that the disk version (on the right), suffers from very low performance numbers and wide standard deviation, permitting few meaningful conclusions. But all schemes show improving throughput through about 4 clients, then remaining steady through 14 clients when performance drops to

about 350 creates/sec. The drop at 15 clients is due to a bottleneck at the single metadata server, and with 3000 files in the directory, there are more seeks and disk synchronization calls that degrade performance for all schemes.

However on tmpfs, until about 5 clients, the pre-creation and leased handles protocols are significantly better than compound, which in turn is better than basecase. This is an effect of the latency differences of the algorithms as presented in the previous set of figures. Beyond 5 clients, timing variations grow larger, but the three new schemes do outperform basecase because they all perform only two *RPCs* unlike basecase that needs three.

5.3 Metadata server scalability

This experiment tests the behavior of the protocols as we move toward more distributed metadata. For the experiment we fixed the number of IO servers at 4, the number

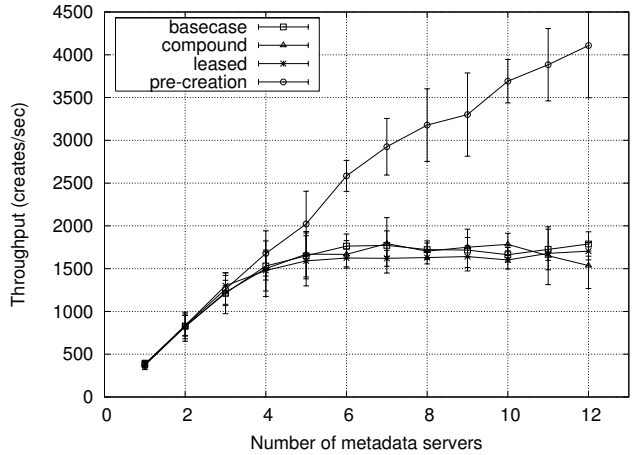
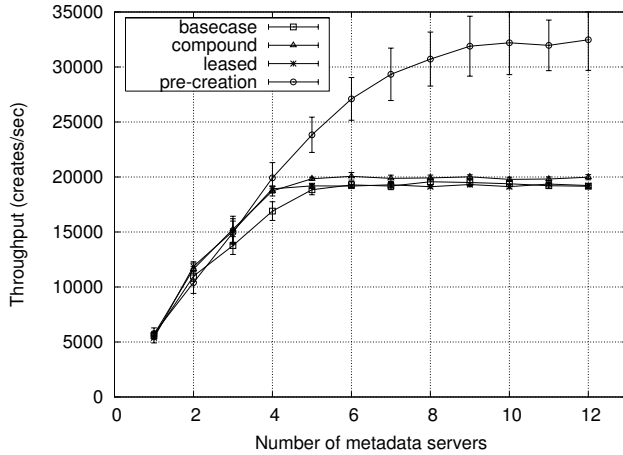


Figure 8. Metadata scalability on (a) tmpfs, (b) disk; different y-axis scales.

of clients at 24 and varied the number of metadata servers from 1 to 12. As file systems grow and as metadata service load increases from increasing numbers of clients, the need to use multiple servers for metadata will become necessary, as will the need to use algorithms that scale well with increasing metadata server counts.

Since PVFS uses a random allocation scheme for new *metafiles*, it was necessary to create a large number of directories to ensure even load across the servers. This was done before the experiment began. The core of the experiment involved the following steps: the clients arrive at a barrier, the leader among the clients starts the timer, each client creates a unique file under a preselected subset of directories, clients arrive at another barrier, the leader stops the timer and finally clients remove the files. We remove the files to maintain constant the size of the metadata database across different runs of the experiment.

Figure 8 shows the metadata scalability on tmpfs and disk, respectively. In contrast to the previous experiments, the patterns in both plots are similar although the tmpfs experiment produced much larger throughput numbers. In both cases, the throughput increases with the number of metadata servers through about 4 metadata servers, at which point all but the pre-creation scheme plateau and remain steady. On the other hand, pre-creation continues to increase, clearly illustrating the scalability difference of that scheme.

When the number of metadata servers is small, as in the experiments of the previous section, the metadata servers were bottlenecks and all four protocols behaved similarly. But beyond 4 metadata servers, the complexity of communicating with IO servers dominates. Metadata operations on disk are slower than tmpfs, but from 5 metadata servers onward, *IO handles* creation time dominates even the overheads from disk access, hence we see 4 metadata servers as

the cutoff in both plots.

Beyond about 9 metadata servers in the tmpfs case, pre-creation begins to flatten off due to network contention effects. Though the number of creates is constant, the rate of creates increases as more metadata servers are added. Given the high rate of creates and the length and number of messages resident in the network, the overall carrying capacity of the single-switch network begins to saturate.

6 Related Work

Distributed file systems like NFS [13], Coda [17] and AFS [11] partition the namespace statically among multiple servers, so file creation is centralized. The pNFS extensions [4] allow for distributed data, but retain centralized metadata.

GPFS [18], GFS [15], Intermezzo [1], Lustre [3] and Frangipani/Petal [22] all use directory locks for creation of files, possibly with a distributed lock management (DLM) infrastructure for better performance. Lustre [3] uses a single metadata server with replication for fault tolerance. In Lustre, clients delegate file creation to the metadata server, which simplifies the create operation. In contrast, PVFS [8] does not rely on a DLM infrastructure but only on isolated per-server locks for atomicity. A truly distributed metadata system such as this breaks a single create operation into a set of multiple operations. Though distributed locking simplifies the create protocol it suffers from scalability issues [9].

Use of compound operations to reduce cost is an old idea [19, 10]. Lazy management of handle leaks in our *datafile* pre-creation protocol is similar in essence to disconnected operations in Coda [7], where the client continues to modify its local cache relying on synchronization with the server for conflict resolution.

The Ceph file system uses dynamic subtree partition-

ing [24] to distribute responsibility for parts of the hierarchy to different servers, and adapts to client access patterns. Due to their hash-driven distribution algorithm that is identical for all files, lists of *datafiles* are not needed in Ceph unlike in more general distributed file systems, obviating any need for our leased handles or pre-creation strategies. Through the use of subtree placement rather than randomly distributed file location, multiple trips to metadata servers are limited.

7 Future Work

PVFS uses Berkeley DB4 for its metadata back-end implementation. For consistency and correctness, metadata changes must be committed to stable storage. Currently PVFS relies on DB4 to regularly commit changes to disk. Since the database has a complex organization, even small changes may translate into a large number of seeks, degrading performance. We hope to improve this situation. Also, we plan to expand our analysis to include other typical metadata operations, such as remove and rename.

8 Conclusion

This paper presented an analysis of the design space for file create in distributed metadata parallel file systems, investigating the complexities involved when multiple metadata and IO servers must participate on each creation while maintaining a consistent view of the overall file system. Our hope is that the work will be useful in future metadata designs, especially with the realization that metadata performance is often a bottleneck, thus motivating the need for distributed metadata.

References

- [1] P. Braam, M. Callahan, and P. Schwan. The InterMezzo filesystem. In *O'Reilly Perl Conference 3.0*, 1999.
- [2] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur. PVFS: A parallel file system for Linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, 2000.
- [3] Cluster File Systems, Inc. Lustre: a scalable high-performance file system, Nov. 2002. <http://www.lustre.org/docs/whitepaper.pdf>.
- [4] G. Goodson, B. Welch, B. Halevy, D. Black, and A. Adamson. NFSv4 pNFS extensions. Technical Report draft-ietf-nfsv4-pnfs-00.txt, IETF, Oct. 2005.
- [5] G. Grider, L. Ward, G. Gibson, R. Ross, R. Haskin, and B. Welch. POSIX I/O extensions workshop, 2005.
- [6] IEEE Standard 1003.1. *Portable Operating System Interface (POSIX)*. IEEE Computer Society, 1996.
- [7] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, Feb. 1992.
- [8] R. Latham, N. Miller, R. Ross, and P. Carns. A next-generation parallel file system for Linux clusters. *LinuxWorld*, 2(1), Jan. 2004.
- [9] R. Latham, R. Ross, and R. Thakur. The impact of file systems on MPI-IO scalability. In *Proceedings of EuroPVM/MPI*, Budapest, Hungary, 2004.
- [10] D. E. Lenoski and W.-D. Weber. *Scalable shared-memory multiprocessing*. Morgan Kaufmann, San Francisco, 1995.
- [11] J. H. Morris, M. Satyanarayanan, M. H. Conner, J. H. Howard, D. S. Rosenthal, and F. D. Smith. Andrew: a distributed personal computing environment. *Communications of the ACM*, 29(3):184–201, 1986.
- [12] D. Nagle, D. Serenyi, and A. Matthews. The Panasas ActiveScale storage cluster—delivering scalable high bandwidth storage. In *Proceedings of the ACM/IEEE SC2004 Conference (SC'04)*, Pittsburgh, PA, Nov. 2004.
- [13] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS version 3: Design and implementation. In *USENIX Summer Technical Conference*, pages 137–152, 1994.
- [14] R. Recio, P. Culley, D. Garcia, J. Hilland, and B. Metzler. An RDMA protocol specification. <http://www.ietf.org/internet-drafts/draft-ietf-rddp-rdmap-04.txt>, Apr. 2005.
- [15] RedHat. Red Hat Global File System (GFS). http://www.redhat.com/en_us/USA/home/solutions/gfs/.
- [16] D. Roselli, J. Lorch, and T. Anderson. A comparison of file system workloads. In *Proceedings of the 2000 USENIX Annual Technical Conference*, pages 41–54, June 2000.
- [17] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, 1990.
- [18] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *First USENIX Conference on File and Storage Technologies (FAST'02)*, pages 231–244, Monterey, CA, Jan. 2002.
- [19] S. Shepler, B. Callaghan, D. Robinson, et al. Network file system (NFS) version 4 protocol. IETF RFC 3530, Apr. 2003.
- [20] E. Smiri, R. A. Aydt, A. A. Chien, and D. A. Reed. I/O requirements of scientific applications: an evolutionary view. In *Proceedings of High-Performance Distributed Computing (HPDC'96)*, pages 49–59, Washington, DC, Aug. 1996.
- [21] R. Thakur, W. Gropp, and E. Lusk. On implementing MPI-IO portably and with high performance. In *Proceedings of the Sixth Workshop on I/O in Parallel and Distributed Systems (IOPADS'99)*, pages 23–32, Atlanta, GA, May 1999.
- [22] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: a scalable distributed file system. In *Proceedings of SOS'97*, pages 224–237, St. Malo, France, Dec. 1997.
- [23] TOP500.Org. Top 500 supercomputer sites. <http://www.top500.org/stats>, 2006.
- [24] S. A. Weil, K. T. Pollack, S. A. Brandt, and E. L. Miller. Dynamic metadata management for petabyte-scale file systems. In *Proceedings of SC'04*, Pittsburgh, PA, Nov. 2004.