# Five Years Of Experiences With Wide-Area User Space File System: SLASH2

Zhihui Zhang, Robert Budden,
J. Ray Scott, Derek Simmel
Pittsburgh Supercomputing Center
{ zhihui, rbudden, scott, dsimmel }@psc.edu

Jared Yanovich
Google
jaredy@google.com

***Abstract*— SLASH2 is an open-source distributed file system that has been in production use at Pittsburgh Supercomputing Center (PSC) for more than five years. Over these years, the development and deployment of SLASH2 have been supported by several multi-million dollar projects funded by National Archives and Records Administration (NARA) and National Science Foundation (NSF). While SLASH2 exemplifies many of the design elements of a state-of-the-art distributed file system (e.g., Lustre and GPFS), it distinguishes itself from these file systems in two important aspects. First, SLASH2 is designed from the outset to be a wide-area file system with features like replication and global mount. Second, SLASH2 operates entirely as a user-mode application, which provides several important advantages compared to typical kernel-mode file systems.**

**In this paper, we discuss the history, the architecture, the deployment, and the performance of SLASH2. Throughout our discussion, we will explain why the two aspects mentioned above make big differences for SLASH2. In a nutshell, we are going to make the case for SLASH2 by sharing the valuable experiences that we have gained over its production operation at PSC and elsewhere. Today, SLASH2 is no longer a proof-of-concept file system. It has established a track record of serving the scientific community.**

***Keywords—wide-area network; distributed file system; user space; open source***

## I. INTRODUCTION

The Pittsburgh Supercomputing Center (PSC) has a long history of developing software to meet the challenges in HPC environments. In the past, PSC used an industrial tape robot to manage its data archives. To reduce the amount of tape movement, a file system called SLASH [6] – scalable lightweight archival storage hierarchy – was developed around 2005. SLASH was basically a disk-based caching system built in front of the tape-based archiving system to reduce its latency. In 2008, PSC developed ZEST [10], a checkpoint data storage system for large supercomputers. PSC was awarded a patent for ZEST, which has inspired further work by other researchers [12]. The experiences gained through these two successful projects have proved to be valuable for the development of SLASH2, the next generation of SLASH, because the core developers of these two projects also worked on the SLASH2 project, which was started in 2009.

The first production use of SLASH2 was to replace the multi-petabyte tape-based archiving system mentioned earlier in 2011 to reduce overall cost and improve performance [14]. These goals were within our reach because the price of disk drives has fallen and there is no software license fee for SLASH2. As part of the project, the tape-based archiving system was actually one of the I/O servers of the first SLASH2 installation until all its data have been migrated to disk-based I/O servers. This enabled continuity of access to the data by users while the migration from tapes to disks was under way. Today, SLASH2 runs on several systems at PSC and elsewhere, including PSC's new Bridges supercomputer, which serves more than 800 clients with several petabytes of storage. The maximum number of files hosted by a single SLASH2 installation to date is well over 200 million. SLASH2 has also seen some limited exposure in the cloud environment [20].

The rest of the paper is organized as follows. Section II reviews the architecture and some details of SLASH2. Section III explains more details in SLASH2 implementation. Section IV discusses the deployment of a SLASH2 file system to show its ease of use. Section V addresses the performance issues of SLASH2. Section VI compares and contrasts SLASH2 with a few well-known distributed file systems. Section VII outlines some future work for SLASH2, and finally we conclude our paper in Section VIII.

## II. ARCHITECTURE

Admittedly, there are many similarities between SLASH2 and other contemporary distributed file systems such as Lustre and GPFS, which are the two leading production file systems used in HPC environments. So our focus here is not to cover every technical detail of SLASH2, but rather to highlight the high-level design choices for SLASH2 that make it unique – a wide-area network (WAN) file system that runs completely in the user space. Readers with distributed file system knowledge will surely find familiar terminologies and techniques in our discussion below.

### *2.1 Overview*

Like many other distributed file systems, SLASH2 is composed of three different kinds of services: metadata server (MDS), I/O server (IOS), and client. A SLASH2 installation must have at least one MDS, one or more I/O servers, and one or more SLASH2 clients.

As its name implies, the MDS is responsible for various metadata operations. To do so, it must store the attributes of files and directories. The MDS also decides which I/O servers should provide the storage for a file. This information is sent to a SLASH2 client upon request so that the SLASH2 client can communicate directly with the I/O servers for read and write operations without further involvement of the MDS. Figure 1 illustrates the relationship between the three services provided by SLASH2:

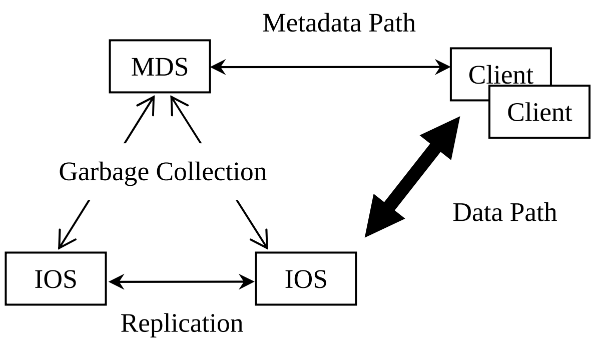

Figure 1 Components of a SLASH2 file system

To start an operation on a file, a SLASH2 client must first communicate with the MDS to retrieve or create its attributes. After that, the SLASH2 client can contact directly with an I/O server chosen by the MDS to read and write data for the file. As an object-based file system, I/O servers are free to use whatever strategies to allocate disk space for the file without any interference from either the MDS or the SLASH2 client. After a write is done, the I/O server should report the file disk usage to the MDS. If the file is later truncated or deleted, the MDS will send a request to the I/O server to reclaim the disk space used by the file. The MDS can also instruct two I/O servers (source and destination) to replicate data blocks of a file on behalf of a user request.

### *2.2 Single Name Space*

In a SLASH2 deployment, the MDS servers, the I/O servers and the SLASH2 clients can be located anywhere in a wide-area network. Nonetheless, all the files and directories in a SLASH2 deployment are accessible from a single name space, which is illustrated in the following Figure 2:

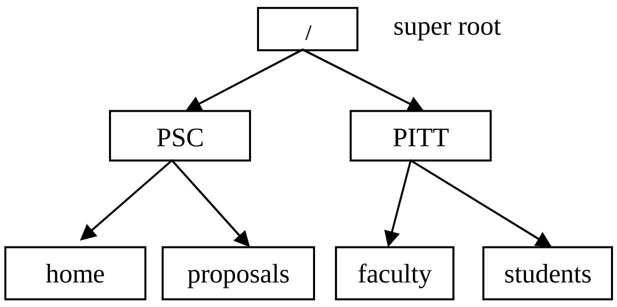

Figure 2 Single namespace of a SLASH2 file system

The namespace shown in Figure 2 is constructed by two MDSes: one MDS is at PSC (Pittsburgh Supercomputing Center) and the other MDS is at PITT (University of Pittsburgh). PSC and PITT happen to be the names used in the configuration files for these two sites. If a SLASH2 client mounts against the MDS at PSC, it will see two directories: *home* and *proposals*, directly under the root. However, if the global mount feature is enabled for the MDS at PSC, the same client will see two directories named after their respective MDSes directly under the so-called super root: *PSC* and *PITT*.

Regardless of whether the global mount feature is enabled, each MDS takes care of its own partition of the global name space. As a SLASH2 client, it can mount against the MDS at either PSC or PITT. If the global mount feature is enabled at the MDS, the SLASH2 client will see the entire name space shown in Figure 2. Otherwise, the client will only see files and directories under either *PSC* or *PITT*.

Incidentally, a SLASH2 installation can be made of an MDS at site A, a couple of I/O servers at sites A and B, and a bunch of clients at site C. So there is no hard requirement that a site must have an MDS. In other words, a SLASH2 installation can span several administration domains.

The idea of global mount originates from the famous Andrew File system [1]. It statically partitions a global name space into different parts that are administered by different organizations. The global mount feature is appealing because it respects the natural administrative boundary between different sites and it is easy to implement with some foresight in place (we will cover this shortly).

During the early stages of the SLASH2 project, we did explore ways to replicate metadata across different MDSes so that a SLASH2 client can communicate with a close MDS instead of a remote one to reduce latency, improve reliability, and eliminate a single point of failure.

Our first idea was to use RPCs to propagate metadata updates between MDSes so that eventually all of them would have a consistent view of the global name space. While this sounds straightforward on paper, it entails a lot of efforts to deal with various corner cases (e.g., adding a new MDS). In addition, as the number of MDSes grows, the number of namespace update RPCs will multiply. Our second idea was to use snapshot send and receive feature supported by modern file systems (e.g., ZFS and Btrfs) to synchronize metadata updates between MDSes. This would require really close collaboration among system administrators working for different organizations even though we probably don't have to write a lot code for it.

Another problem in replicating metadata is that we must assign exclusive ownership of a directory to an MDS so that all updates to the directory will be handled by that MDS to avoid potential conflicts. If an MDS dies, perhaps a distributed consensus protocol (e.g., Paxos [2]) can be used to take away the ownership of the directories managed by the MDS. In the end, we gave up on metadata replication because it is not really useful in practice and it is hard, if not impossible, to get all the corner cases right [18]. Instead, we have chosen to implement the global mount feature to support multiple MDSes in a SLASH2 deployment.

To support global mount, there must be a way to distinguish files managed by different MDSes running at different sites. For this purpose, 10 out of the 64 bits of the file identification number or FID (analogous to the Unix inode number) are used as the site ID, or rather, MDS ID because we can run more than one MDS at the same site. Given the site ID, the client can communicate with the right MDS to read and write a file. Historically, SLASH2 reserves some bits in the FID for purposes other than identifying the site ID, so it only uses 42 bits in FID to distinguish files managed by the same MDS. This situation can probably be improved someday to free up more bits. However, even with 42 bits and a creation rate of 600 files per second, the FID space won't be exhausted after more than 200 years. So SLASH2 can afford to simply assign the next FID for a new file or directory and never reuse a FID.

With global mount, each MDS can be potentially managed by different institutions. There is no need that all or a majority of the MDSes must be up at the same time to reach some kind of consensus. If an MDS is down, only the part of the namespace managed by the down MDS is not accessible. Furthermore, if an MDS crashes, it can be restarted very quickly because it is just a user space process. This is great for system availability because some large shared memory machines can take a very long time (e.g., close to an hour) to reboot based on our experiences. So any kernel crash on such a system means a long down time. Finally, the native file system used by an MDS to store SLASH2 metadata can be built on RAID storage to further improve the reliability of the MDS.

### *2.3 Files and Directories*

At the high level, SLASH2 essentially provides a light-weight service that glues disparate storage resources together. It can be considered as a file system built on top of various native file systems running at each MDS or I/O server. There is no limit on the type of the underlying file system as long as it supports the POSIX file interface (e.g., Ext4, SGI DMF, GPFS, Btrfs, and ZFS). Therefore, SLASH2 does not displace any existing storage solutions. It just makes them more readily accessible.

Accordingly, a SLASH2 file is implemented as one or more component files in the underlying native file systems running on an MDS or an I/O server. For a SLASH2 file, a component file always exists in the file system running on an MDS. The component file on the MDS stores various attributes of the corresponding SLASH2 file, including the storage map or directory contents of the corresponding SLASH2 file depending on whether the SLASH2 file is a regular file or a directory. The name of the component file on the MDS is the same as the name of the corresponding SLASH2 file. The following Figure 3 shows the format of the component file for a regular SLASH2 file:

| | | | |
|---|---|---|---|
| Inode | Block map 0 | Block map 1 | … |

Figure 3 Storage map of a regular SLASH2 file

The inode contains a list of at most 64 I/O servers where the file data can be stored. A regular SLASH2 file is broken up into 128MiB blocks. There is a block map for each potential block in the regular SLASH2 file. The block map maintains the state of the block on each of the I/O servers listed in the inode. The state of a block can be valid, garbage, pending replication, pending truncation, etc.

If an I/O server stores one or more blocks for a SLASH2 file, it has to create a component file for the SLASH2 file on its native file system. The name of a component file on an I/O server is the concatenation of the FID of the SLASH2 file and a generation number, separated by an underscore symbol. The generation number increments whenever its corresponding SLASH2 file is fully truncated (i.e., truncate to zero length). It allows a SLASH2 client to write new data into a fully truncated file without waiting for the garbage collection to complete first on the file. We will see an example of the name of a component file near the end of Section III.

The block is the unit of storage allocation at the I/O server level from the point of view of an MDS. This means that a 128MiB block must exist in its entirety on a single I/O server. However, SLASH2 allows any block in any file to be replicated on multiple I/O servers. So a SLASH2 client can choose which I/O server to read file data from. Note that if a block on an I/O server is updated, all its other replicas, if any, are invalidated automatically by the MDS.

This means that the block is also the unit of data coherence management. Following the standard practice employed by many distributed file systems, a SLASH2 client must request a read or write lease for a block from the MDS before it can ask the I/O server to read and write the block. If two clients happen to write to the same block within a file, both clients must fall back to the non-caching (we call it *direct I/O*) mode to perform I/O. Of course, a lease is tagged with a timestamp and a sequence number so that an I/O server can detect obsolete leases easily.

## III. IMPLEMENTATION

In this section, we are going to explore more implementation details of the three services of SLASH2 – MDS, I/O server, and client.

### *3.1 The Metadata Server (MDS)*

In SLASH2, as in other distributed file systems, the MDS is the brain of the file system. It performs the following tasks in SLASH2:

- Namespace management. All namespace operations (e.g., creating a file, deleting a file, looking up a file, reading a directory) must go through an MDS. The MDS also store attributes for all the files in a SLASH2 deployment such as the file size, modification time, etc.

- Storage management. The MDS stores the block map for all regular files, which means that a SLASH2 client must talk to a MDS before reading and writing any file. The MDS also decides which I/O server should receive new data based on the storage usage on each I/O server and client preferences.

- Lease management. In addition to handing out the storage map of a file upon request, the MDS also grants read or write leases to the requesting SLASH2 client. The lease is a contract between a client and an MDS that specifies which I/O server that the client should contact for file I/O and for how long. Leases are managed on a per-block and per-file basis.

- Garbage collection. Whenever one or more block in a file is invalidated, the MDS will communicate with the relevant I/O servers to reclaim some storage blocks occupied by the file. Block invalidation can occur when a file is truncated or deleted. It can also happen because a more recent copy of the block has been stored on a different I/O server. The MDS must hold onto pending garbage collection requests because an I/O server may be unavailable for a while.

- Replication Management. A user can request to replicate blocks in a file to a specified I/O server. The replication engine in the MDS selects a source I/O server and asks the target I/O server to retrieve the file block directly from there. When the replication is done, the result is reported back to the MDS and the block map of the file is updated accordingly.

Overall, an MDS acts as a coordinator that does its best to delegate as much work as possible to other parties. In particular, it is not involved in the file data I/O path. To further reduce the load on an MDS, a SLASH2 client caches directory contents and leases provided by an MDS as long as possible. These strategies are also shared by many other distributed file systems.
However, a key feature in SLASH2 that does not necessarily exist, at least in the same form or shape, in other distributed file systems is replication. This feature is designed to allow a user to have complete control of where to store his or her data among available I/O servers. It can be leveraged for various purposes:

- Local Performance: A client can arrange for its data to be replicated to a set of I/O servers that are geographically closer than other I/O servers for better performance. This can reduce the latency of I/O operations when SLSAH2 is used in a wide-area network.

- Data Protection: While each I/O server should run on a native file system with built-in RAID support, replication can add another dimension of protection against data loss. If a copy of a block is not available or is corrupted on an I/O server, it can be retrieved from another I/O server.

- Data Migration: If all data blocks on an I/O server have been replicated to other I/O servers, then that I/O server can be removed from a SLASH2 deployment and re-purposed.

The replication feature is very flexible and is completely driven by a user. A user can replicate a single file in its entirety. Or the user can replicate certain blocks within a file. A replica of a block can also be added or removed at any time. The target I/O servers are selected by the user as well. Replication policies can be set on a directory and inherited by new files created in that directory. Note that the use of replication is optional. Even if all blocks in a file have exactly one valid copy, they can be spread across multiple I/O servers for better aggregate throughput.

Now let us move on to some key implementation details. The file system used by the MDS is based on ZFS-fuse 0.7.0 beta, which is a user space implementation of ZFS. We chose it so that we can easily back up the MDS file system with snapshots for accounting and disaster recovery purposes. Over time, we have made the following changes to our ZFS-fuse fork:

(1) Transaction group: The ZFS code uses transaction groups to commit pending changes atomically. For an

MDS, it uses an external journal on a file or a device to log changes to the name space (e.g., create a file) before replying to a client. If the MDS crashes, the log entries can be replayed. However, it must make sure that the corresponding action has not already been taken by the ZFS before replaying. To do so, each log entry in the journal is tagged with a transaction group ID in which the corresponding changes in the ZFS should happen. In addition, there is a special cursor file in the ZFS that records the ID of the last committed transaction group. The transaction that updates the cursor file must be the first one to start a new transaction group and the last one to commit in the same group.

(2) FID name space: As we have discussed earlier, there is a component file or directory in the file system used by the MDS for each SLASH2 file or directory. In fact, the hierarchy of the component files in the native file system used the MDS matches the hierarchy of the SLASH2 name space seen by a SLASH2 client. In addition to component files, the native file system used by the MDS also contains other internal files (e.g., the cursor file mentioned above). They are not visible from a SLASH2 client at all.

As a distributed file system, SLASH2 needs to operate on FIDs instead of file names. For example, the file creation RPC presents the FID of the parent and the name of the file to create. To be able to look up a SLASH2 file or directory by its FID, a hard link to its corresponding component file or directory is created. The name of the hard link and its location is based on the FID of the target file or directory. Collectively, all these FID hard links constitutes the so-called FID name space. The FID name space is only used internally by an MDS.

At this point, some astute readers might have noticed that the MDS creates hard links for directories, not just for regular files. This practice can cause confusion because it allows a file or directory to have more than one parent. Fortunately, it only creates a forward link from an FID name to its corresponding directory name, so no directory cycle actually exists.

We are considering using symbolic links instead of hard links to construct the FID name space. That way, system administrators will be able to back up and restore all metadata with their familiar tools (e.g., *rsync*) without solely relying on the snapshot feature. There might be some small performance loss, but the gain of manageability will be appreciated for sure.

Incidentally, the native file system used by the MDS is only visible to the MDS when it is running. There is no particular reason for this – it is just an implementation detail. However, when the MDS is not running, we can use FUSE to mount its file system. In fact, that is the way in which the SLASH2 tool *slmkfs* populates the initial contents of the file system.

(3) Bug fixes. One particular type of bugs was in the memory management area of ZFS-fuse, which uses mmap() to mimic kernel space allocations performed by a kernel-resident version of ZFS. Historically, we encountered mysterious crashes whenever the MDS uses more than 32GiB of physical memory. Luckily, these incidents are in the history book now.

In retrospect, choosing ZFS-fuse was a bit risky because it is not maintained any more even though ZFS was already touted as an industrial-strength file system at that time. We should probably write a user-space MDS based on a kernel-resident file system that supports snapshots (e.g., ZFS-on-Linux and Btrfs) even if doing so might take some performance hit. If so, system administrators would be able to back up SLASH2 metadata using well-known Unix tools because all the files are naturally accessible by the MDS and other user space tools at the same time. Furthermore, we can easily move the MDS to a different file system instead of sticking with one. Anyway, ZFS-fuse has worked out so far for us.

The user-space MDS has some great benefits. First, the MDS is run under the control of a simple wrapper script. If the MDS crashes, the wrapper script will collect its core file, send an email to system administrators, and restart the MDS. This kind of automation is very easy to accomplish when the MDS runs in the user space. Furthermore, if a kernel-space MDS crashes, it likely brings down the entire system. On some big machines, a reboot literally takes almost an hour. As a user-space MDS, it can restart quickly after a crash because it is just a regular process. This allows a client to delay and retry its operations for a reasonable time instead of returning a failure on the first sign of trouble.

Files in the file system used by the MDS internally are generally small because they only store the attributes, the storage map, and the directory contents for corresponding SLASH2 files or directories. While a SLASH2 client does have name cache and readdir-plus like support, it is highly recommended to use SSD as the backend storage for ZFS-fuse and the MDS journal. Doing so will greatly improve the performance of namespace operations.

Historically, we have configured the MDS to run in the failover mode. The storage for the file system used by the MDS is dual-ported. A backup MDS can take over the storage when the primary MDS is down. This configuration was never really used and thus we can no longer justify the extra cost for doing so.

### *3.2 The I/O Server (IOS)*

An MDS has a set of at most 65,536 I/O servers by default to store actual file data. An I/O server can run on top of any file system that provides the POSIX file interface. In fact, all data used by an I/O server are stored under one regular directory. The underlying file system that provides the directory for SLASH2 can be a local file system (e.g., Ext4 and ZFS-on-Linux) or a distributed file system (e.g., Lustre).

The responsibilities of an I/O server are straightforward. First, it has to serve read and write requests from a SLASH2 client. Second, it has to serve a read request from a peer I/O server for replication purpose. Third, it has to process garbage collection requests from its MDS. Finally, it has to report to its MDS on its storage usage.

The I/O server is designed to be stateless, so that it can recover from a crash on its own. The data for a SLASH2 file is stored in a native file whose name and location are determined by the FID of the SLASH2 file. The directory layout of these native files is similar to the layout of the FID name space created by the MDS. However, no hard links are used. We will peek into the layout of the FID name space used by an I/O server near the end of Section III.

The I/O server is the simplest of the SLASH2 services. All it does is to read and write files upon the request of a client or another I/O server provided that the lease associated with the request has not expired. The I/O server also maintains a *sliver cache* for better I/O performance. One sliver is 1MiB worth of file data. So there are 128 slivers per block.

An I/O server can refuse to accept new data when that its storage usage has exceeded a certain limit (say 95% full). This is important because file system performance tends to drop when it becomes full. On the other hand, an MDS can be instructed to stop granting new write leases targeted to an I/O server. Both behaviors can be tweaked on the fly by a system administrator.

We recommend that an I/O server runs on a file system that supports punch-hole and seek-hole capabilities (e.g., Ext4). If a block in the middle of a file is invalidated, the storage used by the block alone can be reclaimed independently. When the storage usage on an I/O server has exceeded its current limit, existing blocks can still be overwritten although an attempt to write a new block will be rejected with ENOSPC. In other words, writing to a hole is not allowed when storage is tight.

### *3.3 The Client*

The SLASH2 client is a multi-threaded program that presents a POSIX interface to applications by mounting against an MDS. It is implemented as a FUSE file system [21]. Currently, it uses the low-level interface provided by the standard FUSE library.

The core logic of the SLASH2 client is actually encapsulated into a shared library that must be loaded into a generic file system client framework. As a result, we can unload the shared library at run time (after flushing all on-going operations) and reload a new version of the library. This opens the door for stackable processing and seamless upgrade in the future.

Historically, a FUSE file system is frowned upon because it incurs extra memory copy and context switch overhead. People have also argued that a file system must live in the kernel space [3].

While we were fully aware of these arguments, we still chose to write a FUSE file system because user-space file system development is much more cost-effective. And we don’t have to worry about compatibility issues each time a new kernel release is out. During early stages of SLASH2 deployment, we often attached the GNU debugger *gdb* to a running SLASH2 daemon to diagnose on the fly in a production environment with minimal intrusion within customer impact obligations. If SLASH2 client was implemented in the kernel, this practice would be simply impossible.

Just because SLASH2 is implemented in the user space, does not mean it is easy. SLASH2 is not a simple pass-through FUSE file system. It has to handle all the jobs required for a sophisticated distributed file system client including sending and receiving RPCs, cache management, permission checks, etc. To illustrate our point, we are going to share some experiences below.

In order to improve performance, a SLASH2 client allows a FUSE thread to return as soon as an RPC is sent out. In general, we assume that one I/O request can be satisfied by one RPC request. This is not necessarily true. In addition, the replies for multiple RPCs issued on behalf of the same I/O request can arrive out-of-order. Therefore, a SLASH2 client can only send a reply to a FUSE request after all its RPCs have completed. As a result, the I/O request control flow is more complicated than it may appear to be.

As a user space file system, SLASH2 can take advantages of libraries and even utilities (e.g., ZFS tools) in the user space easily. However, we have learned to be careful sometimes. The SLASH2 client maintains its own page cache. One page is 32KiB. As we have found, at least some versions of GNU C library (glibc) [22] does not work well if allocations of different sizes are mixed together, despite our attempts to tweak various glibc parameters. As a result, our system administrators complained that a SLASH2 client consumed 2GiB of physical memory even if it had been idle for a while. This problem has been fixed by using mmap() to allocate large page cache memory separately from the rest of small allocations.

Finally, to make SLASH2 friendly in a wide-area network, a SLASH2 client is also equipped with the following features:

- Delay and retry: A SLASH2 client stands ready to delay and retry its RPCs to ride out network glitches and server crashes. In addition, a SLASH2 client will not send out more RPC requests when there are already too many outstanding requests. The behavior of delay-and-retry and the RPC throttling mechanism can be tweaked with several runtime parameters.

  While these measures sound simple, they turn out to be effective in improving the stability of a SLASH2 installation on a large system. Imagine all 800 clients read and write files simultaneously that are stored on a bunch of I/O servers. Chances are that one of these I/O servers can be swamped and become not responsive. Incidentally, if an I/O server is down, a SLASH2 client can ask the MDS to pick a different one, abiding coherency semantics.

- MDS routing: A SLASH2 client is able to communicate with the right MDS based on the site ID embedded in the FID of a file or a directory. This is used to support the global mount feature exported by an MDS.

- UID/GID mapping: A SLASH2 client can provide mapping between local and remote credentials to allow access to files from a client that lives outside of the administrative domain of the MDS.

Incidentally, when SLASH2 was first tested in a wide-area network, we encountered many mysterious issues with the network including connection drop, firewall blockage, MTU mismatch, etc. Through these experiences, we now check these types of issues first when a user reports a problem.

### *3.4 Putting Everything Together*

All three SLASH2 services (MDS, I/O server, and client) communicate with each other via RPCs using TCP/IP sockets. For better performance, we have also implemented a batch RPC mechanism that can be used for certain types of RPCs (i.e., replication request and garbage reclamation) between an MDS and an I/O server.

To ensure RPC messages are not corrupted or tampered with for any reason, a secret key is securely shared in advance among participating SLASH2 services. The contents of the secret key are pre-appended to each RPC message for the purpose of calculating the message digest for the RPC message. Services receiving a RPC message must calculate the expected message digest over the contents of their copy of the shared secret key and the RPC message. If the result matches the message digest tagged at the end of the RPC message, the RPC message is validated.

The identities, locations, and other information of the MDS and its I/O servers are described by a text configuration file that is shared by all services. To avoid a single point of failure, all services in SLASH2 can be started or shut down at any time in any order. However, they do exchange periodic heartbeat RPCs to monitor the status of other services.

The following Figure 4 shows an example configuration file for a small SLASH2 development setup in which all resources live at the same site named PSC:

```
set port=1000;
set net=tcp10;
set pref_mds="orange@PSC";
site @PSC {
        site_id             = 0x123;
        site_desc           = "SLASH2 @site PSC";
        fsuuid              = 0x1234567812345678;
        resource orange {
                desc        = "PSC MDS orange";
                type        = mds;
                id          = 0x11;
                nids        = 128.182.99.28;
                journal   = /dev/sdb;   }
        resource lime {
                desc        = "PSC IOS lime";
                type        = standalone_fs;
                id          = 0x22;
                nids        = 128.182.99.27;
                fsroot    = /local/lime/zhihui-s2;
        }
        resource lemon {
                desc        = "PSC IOS lemon";
                type        = standalone_fs;
                id          = 0x25;
                nids        = 128.182.99.26;
                fsroot   = /local/lemon/zhihui-s2;
        }
        resource allios {
                desc        = "all I/O servers";
                type        = cluster_noshare_lfs;
                id          = 0x1fff;
                ios         = lime, lemon;
        }
}
```

Figure 4 Example SLASH2 configuration file

In the above example, the host *orange* is the MDS. It has two physical I/O servers: *lime* and *lemon*. The two I/O servers store their data under directories rooted at */local/lime/zhihui-s2* and */local/lemon/zhihui-s2* respectively. The configuration file also specifies a logical I/O server named *allios*. If a SLASH2 client specifies *allios* as its preferred I/O server, the MDS will try to spread write operations coming from the client among *lime* and *lemon* by picking one of them randomly.

Now let us peek into the FID namespace on an I/O server by looking up a component file stored there. The corresponding SLASH2 file for the component file is named *myfile.txt* as shown in the following Figure 5:

```
On a SLASH2 client named yuzu:

$ zhihui@yuzu: /zzh-slash2/zhihui$ ls -il myfile.txt
327636872901899687 -rw-rw-r-- 1 zhihui root 298 Feb 17 15:59 myfile.txt
$ zhihui@yuzu: /zzh-slash2/zhihui$ printf "%x\n", 327636872901899687
48c000000a335a7

On the I/O server named lemon:

# bash-4.2# pwd
/local/lemon/zhihui-s2/.slmd/1234567812345678/fidns/0/a/3/3
# bash-4.2# ls -ali 048c000000a335a7_0
112918821 -rw------- 1 root root 298 Feb 17 15:59 048c000000a335a7_0
```

Figure 5 SLASH2 component file location

As shown in Figure 5 above, the FID of file *myfile.txt* is 0x48c000000a335a7. The 10 most significant bits of the FID are 0100100011 or 0x123, the site ID of PSC as defined in the configuration file shown in Figure 4. The directory of the component file for *myfile.txt* is determined mostly by concatenating the root directory of the I/O server, the UUID of the file system, and 4 middle hexadecimal digits (e.g., 0a33) of the FID, which are used to navigate through the FID namespace. In our example, the contents of *myfile.txt* and its component file named 048c000000a335a7_0 are identical because the SLASH2 file fits within one 128 MiB block. The file has never been fully truncated, so its generation number suffix remains 0.

The above example also shows that all component files on an I/O server live under one directory in the native file system. If we need to move an I/O server manually from one node to another node, all we have to do is to copy over its directory and change the address of the I/O server in the configuration file. Of course, we can also leverage the SLASH2 replication engine to replicate data elsewhere and then invalidate all blocks of all files on the I/O server to be retired.

## IV. DEPLOYMENT

Since 2011, SLASH2 has been deployed on several systems at PSC and elsewhere. These experiences, sometimes painful, have been valuable in improving the maintainability and the manageability of SLASH2.

### *4.1 System Configuration*

We have typically used commodity hardware to run SLASH2 servers and clients. On the MDS side, we configure the ZFS pool to be either two-way or three-way mirrored. On the I/O server, we use JBOD configured as a RAID array (e.g., 8+3 ZFS RAIDZ3). In terms of operating system software, we have used FreeBSD and various Linux distributions including RHEL/CentOS and Ubuntu. We initially chose FreeBSD simply because ZFS was only available on FreeBSD at that time.

As expected, we have not encountered any compatibility issues because SLASH2 runs entirely in the user space and the POSIX file system API between the user space and the kernel space has remained stable. This reduces system administration hassles because we don't have to download kernel patches to upgrade our file system anymore. SLASH2 is an open source project. All its source code files are available on the GitHub. As of this writing, SLASH2 has more than 104,000 lines of source code.

To install SLASH2, one can retrieve the current stable SLASH2 release, then build and install it as follows:

```
$ git clone https://github.com/pscedu/slash2-stable
$ cd slash2-stable
$ make && make install
```

### *4.2 SLASH2 Monitor Scripts*

A SLASH2 deployment usually has multiple I/O servers and clients. For easy management, we have developed the following three monitor scripts: *slashd.sh*, *sliod.sh*, and *mount_slash.sh*. These are handy wrapper scripts that can be used to launch an MDS, an I/O server, and a client respectively. A monitor script usually has nothing to do afterwards. However, if a service crashes, a monitor script restarts the service, saves the core dump file, and sends a crash report email to developers. A monitor script can be started as follows:

```
# slashd.sh –dvg
# sliod.sh
# mount_slash.sh –P profile
```

As we can see, a monitor script can run by itself without arguments. It can also accept some options: *-d* for daemonize, *-v* for verbose, and *-g* to run the service under the GNU debugger *gdb*. A profile can be used to set up the environment for a SLASH2 installation. A profile specifies the location of SLASH2 binaries, the pathname of the log file, various runtime options, and other environment settings. If a profile is not given, reasonable system defaults are used. A profile is especially useful when a SLASH2 client machine is shared by multiple SLASH2 instances.

### *4.3 SLASH2 Specific Tools*

All SLASH2 services run as a regular process on a machine. Therefore, all traditional Unix tools such as *top*, *kill*, *ps*, etc. can be used to manage them. In addition, SLASH2 provides three custom tools *slmctl*, *slictl*, and *msctl* to manage the MDS, the I/O server, and the SLASH2 client respectively.

These tools have the same look and feel and are very easy to use. They talk to their respective services through a local Unix socket. These tools can be used to check system status, collect I/O statistics, turn on and off some features, and much more. Man pages for these tools are available as well. The following are few examples of how to run these tools (outputs have been tweaked to fit the format of this paper):

```
$ slmctl -p sys | tail -2
sys.uptime                          23d22h51m
sys.version                         42677
```

The above example shows that the MDS service has been up for more than 23 days and the software version is 42677, which is the number of *git* commits.

```
bash-4.2# ./msctl -s connection
resource    host        type    flags stkvers txcr  #ref     uptime
==============================================
PSC
  Orange orange.psc.edu mds -O-   42670    8     1    6d20h25m
  lemon lemon.psc.edu local  -O-   42677    8     1    4d01h28m
  lime    lime.psc.edu  local   -O-    42670    8     1    6d20h25m
```

The above command shows the service connection status from a client point of view. It is connected to an MDS and two I/O servers, all of them are at the site named PSC. Note that services do not have to run at the same version number, as long as they are compatible with each other in terms of RPC protocol. Also, they don't have to be brought online at the same time.

```
$ msctl –sop | grep peer
opstat                   avg rate      max rate      cur rate           total
============================================
peer-192.231.243.100   350B/s  10.9K/s          0B/s    196.4K
peer-192.231.243.101 38.7M/s 53.7M/s     30.7M/s        1.6G
```

The above command shows the peer connection I/O statistics for a client point of view, including the total number bytes that have been transferred and three data transfer rates. The average rate is a running average rate, so it will go to zero after the connection is idle for a while.

Last, but not the least, these tools have been integrated into the infrastructure at PSC to monitor the health of a SLASH2 file system.

## V. PERFORMANCE

One of the first goals of SLASH2 development was to replace a tape-based archiving system and get better performance. It was not a high bar to achieve because disks are much faster than tapes. And we did that. After that, our main efforts were on improving the stability and manageability of SLASH2. So there is still a lot of work to be done in terms of performance. Nonetheless, we want to share our experiences and thoughts in this area as well.

At the design level, we have made some choices that could affect the performance of SLSAH2 in a positive or a negative way:

- First, the FUSE client may slow things down for some workloads due to extra memory copy and context switch overhead. However, given the fact that SLASH2 is targeted to wide-area use, this overhead is probably not the biggest concern along the entire I/O path. If fact, many other factors, such as the network performance, the number of spindles at the I/O server, or where SSD drives are used at the MDS, can play a bigger role. To date, attention to these areas has produced many SLASH2 performance gains for PSC's HPC/cluster environment.

  If the extra memory copy operations incurred by FUSE become a bottleneck in the future, we plan to explore the zero-copy feature available in modern FUSE implementations. This feature has already been used by some researchers with promising results [17]. We are not worried about context switch overhead because CPUs are much faster than any other major components in the system and using larger I/Os can lessen this overhead.

- Second, SLASH2 allows a user to replicate his or her files to a set of I/O servers that are close to his or her SLASH2 client. This can achieve what we call local performance in a wide-area network.

At the implementation level, we have made some significant efforts to improve the performance of SLASH2 as well:

- The I/O stack. The I/O stack in a SLASH2 client is not trivial because it has to handle all kinds of I/O requests: direct I/O, cached I/O, and asynchronous I/O (AIO). AIO allows a slow I/O server to return an I/O request asynchronously. Of course, the I/O stack also performs read-ahead and write-behind for better performance.

  One of the biggest issues we have overcome is the dismal performance of GeneTorrent with SLASH2. This application is heavily used by some users to download data into SLASH2. GeneTorrent uses multiple threads to write to different parts of the same file simultaneously and these parts are not necessarily aligned on a page boundary. This did not work well with the old SLASH2 I/O stack. The solution is to remember which area in a page is valid because it has just been written instead of always relying on the I/O server to provide the most recent contents. This solution entailed a major code rewrite, but it did bring significant performance improvement for GeneTorrent. Unfortunately, it also introduced some subtle data corruption bugs that took us a long time to understand and fix.

- Name cache. The SLASH2 client maintains a name cache to improve the performance of name space operations. This is the common practice for every file system. In addition, it also uses read-ahead on directory contents (not file data). This aggressive algorithm poses some interesting race conditions that probably prevented us from compiling Linux kernel and GCC source code on a SLASH2 file system for a long time in the early days. In retrospect, the directory read-ahead algorithm helps. However, as it turns out, the biggest boost of performance comes from replacing spinning disks on the MDS with SSD.

The lesson here is that fast and affordable hardware can often make a greater and sooner difference in terms of performance than perfecting a clever algorithm, which is hard to get right due to various race conditions or other implementation challenges.

Based on our experiences, we don’t see why SLASH2 cannot be further improved to perform comparably with kernel-based distributed file systems. As of this writing, the overall performance of SLASH2 has been sufficient for its intended purposes where it is deployed.

## VI. RELATED WORK

In this section, we are going to review several distributed file systems in light of some of the design decisions we have made for SLASH2.

GPFS [4] is a parallel, shared-disk file system developed by IBM. The goal of GPFS is to extend a local file system into a cluster environment for better scalability. All nodes in a GPFS cluster access the same set of disks. They can perform metadata as well as data operations. This means that these nodes must synchronize with each other to avoid data and metadata corruption. In fact, one of the biggest challenges and achievements in GPFS is to scale its locking algorithms for different types of data among competing nodes. At its heyday, GPFS was a breakthrough in terms of scalability. Today, the technology trend has changed. In particular, the prevalent use of object-based storage has obviated the need for a file system node to understand on-disk data structures such as inodes and indirect blocks. In SLASH2, the storage contributed by an I/O server is independently managed. Its capacity can change on the fly without consent from another server. In addition, an I/O server in SLASH2 can be located anywhere in the network and can be added or removed at any time. A centralized storage allocation scheme like the one used in GPFS is no longer practical.

Panache [13] is a project that attempts to make a GPFS cluster globally accessible while maintaining the performance that is available on a cluster file system. Panache uses a classic caching technique to overcome latency and reliability issues associated with a wide-area network. Basically, a remote cluster and a home cluster must coordinate with each other via one or more gateway nodes. All data and metadata must first be cached *persistently* on the home cluster before they are available locally. This means that if a researcher wants to access a single file, he or she has to set up a full-fledged local cluster and replicate at least part of the name space that leads to the file before the file can be accessed. In contrast, SLASH2 can simply replicate the file explicitly to a local I/O server. After that, the local I/O server has the most authoritative copy of the file. There is no need to use RPCs to resynchronize with the master copy because there is no fixed master copy in SLASH2. This strategy works because file sharing is not really common based on our experiences and others [7].

Lustre [5, 15] is the most scalable and popular open source file system used in HPC environments today. The main idea of Lustre is to separate metadata operations from actual data I/O operations. This idea is neither new nor unique. In fact, some commercial parallel file systems such as PanFS [8] and BeeGFS [16] also adopted the same idea. The architecture of SLASH2 is very similar to that of Lustre. In fact, the RPC layer used by SLASH2 is the LNET stack operating in the user space. Compared to SLASH2, Lustre requires kernel modules, which means much higher development and maintenance cost when compared to SLASH2. Lustre also does not give a user full control of where to replicate his or her files. It does not support global mount either. Furthermore, different I/O servers can run on different file systems (e.g., SGI DMF, Ext4, Btrfs, ZFS-on-Linux) in the same SLASH2 installation. No special filter module needs to be written for any of these underlying file systems.

XtreemFS [9] is an object-oriented storage solution for grid data management. Like SLASH2, the developers of XtreemFS also recognized the value of using the file system interface for grid data management instead of relying on GridFTP data transfer tools. However, their approach to keep replicated data consistent is different from SLASH2. Each file is striped across an immutable set of storage servers. On top of that, a complicated protocol between storage servers is required to deal with holes in a file and changes of the file size. As they have mentioned in the paper, any distributed consensus protocol is inherently expensive and does not scale well. And we would add that it is difficult to debug a distributed protocol, which in turn affects the stability of the file system [18]. In SLASH2, storage servers are dumb and stateless. It does not even have any idea of whether its copy of a block is valid or not. In addition, the number of replicas can change on the fly in SLASH2.

CalvinFS [19] is distributed file system that aims to solve the metadata scalability problem by replicating metadata as well as data. Our concerns about their work are as follows. First, distributed protocols are expensive. On top of that, if a consensus cannot be reached because some server is down or swamped, an operation might be stalled. Second, the append-only strategy will generate a lot of garbage. Our experience has shown that users’ appetite for storage can grow in spurts.

In addition, when the amount of free space dwindles, the performance of a file system plummets rapidly. Unfortunately, no detailed discussion of garbage collection is found in the paper. Third, the use of global log storage also concerns us. SLASH2 is designed to bring independently operated storage together. Making a distributed file system component dependent on another globally shared storage will not scale and likely increase system administrative overhead.

## VII. FUTURE WORK

Although SLASH2 has been successfully employed in production for 5+ years, we still consider it a work in progress. Surprisingly, it seems to have caused less trouble than the Lustre installation on a similar scale at PSC based on the number of reports received by our custom support team. In the future, we want to add more features to SLASH2 based on its existing infrastructure. Features like subtree mount, Kerberos authentication are nice to have. Performance wise, we want to leverage the write-back feature of FUSE file system. It is also possible to implement the idea of localized directories on a SLASH2 client [7]. The I/O server can be expanded to access files in a cloud-based storage. Another work item is to complete the export and import feature that obviously can be used to transfer files between a SLASH2 file system and a non-SLASH2 file system. We anticipate that we will continue to develop and improve SLASH2 for many years to come.

## VIII. CONCLUSION

In this paper, we have covered the design and implementation of SLASH2, with observations drawn from 5+ years of operating and improving SLASH2 in production HPC environments. We believe that SLASH2 deserves its place among other commercial and open-source distributed file system alternatives based on our experiences in its development as well as deployment.

In particular, SLASH2 has demonstrated its value in the scientific computing community because of its ability to logically bind existing storage systems – of different types and vendors – into a single POSIX file system. With SLASH2, a user can use the familiar file system interface to manage his or her data without learning specialized tools. In addition, by implementing everything in the user space, the cost of development and deployment is cut down significantly. This is critical because *human cycles* are becoming more valuable than *machine cycles* these days as commodity hardware becomes cheaper.

Obviously, we don’t have anywhere near the financial and technical backing enjoyed by other file system projects. As a result, we have to selectively implement features that are most useful in practice. However, the very fact that we have achieved a lot of success with limited resources confirms the technical choices we have made are largely sound.

In the long run, we hope SLASH2 will continue to mature and grow in terms of features and adoption. We would also like to hear from anyone who is interested in using or working with us on SLASH2 in the future.

## IX. ACKNOWLEDGEMENTS

File system development is a major effort that involves not only developers, but also system administrators and users. First, the authors want to thank NARA and NFS for their support. Second, the authors want to thank the system administrators and users at PSC for their patience and dedication to SLASH2 over the years. No pains, no gains. Early on, SLASH2 used to crash two or three times a week. Now it can run for months without a crash or any signs of memory leaks. We used to take a day to upgrade SLASH2. Now we can finish an upgrade in a few hours. Last, but not the least, we are grateful for the detailed reviews by Kathy Benninger and Beth Lynn Eicher on early drafts of this paper.

## X. REFERENCES

1. John H Howard, An Overview of the Andrew File system, Proceedings of the USENIX Winter Technical Conference, Dallas, February 1988.

2. Leslie Lamport, The part-time parliament, ACM Transactions on Computer Systems (TOCS), Volume 16 Issue 2, May 1998.

3. Brent Welch, The File system Belongs in the Kernel, Proceedings of the 2nd USENIX Mach Symposium, Monterey, November 1991.

4. Frank Schmuck and Roger Haskin, GPFS: A Shared-Disk File system for Large Computing Clusters, Proceedings of the FAST 2002 Conference on File and Storage Technologies, Monterey, January 2002.

5. Philip Schwan, Lustre: Building a File system for 1,000-node Clusters, Proceedings of the Linux Symposium, Ottawa, Ontario, July 2003.

6. Paul Nowoczynski, Nathan Stone, Jason Sommerfield, Bryon Gill, J. Ray Scott, Slash – The Scalable Lightweight Archival Storage Hierarchy, Proceedings of the 22nd IEEE/13th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST’05), Monterey, April 2005.

7. Edward Walker, A distributed file system for a wide-area high performance computing infrastructure, WORLDS'06

Proceedings of the 3rd conference on USENIX Workshop on Real, Large Distributed Systems, Seattle, November 2006.

8. Brent Welch, Marc Unangst, Zainul Abbasi, Garth Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou, Scalable Performance of the Panasas Parallel File system, The 6th USENIX Conference on File and Storage Technologies (FAST '08), San Jose, California, February 2008.

9. Jan Stender, Björn Kolbeck, Felix Hupfeld, Eugenio Cesario, Erich Focht, Matthias Hess, Jesús Malo, Jonathan Martí, Striping without Sacrifices: Maintaining POSIX Semantics in a Parallel File system, 1st USENIX Workshop on Large-Scale Computing (LASCO '08), Boston, June 2008.

10. Paul Nowoczynski, Nathan Stone, Jared Yanovich, Jason Sommerfield, Zest - Checkpoint Storage System for Large Supercomputers, 3rd Petascale Data Storage Workshop, Austin, November 2008.

11. Feiyi Wang, Sarp Oral, Galen Shipman, Oleg Drokin, Tom Wang, and Isaac Huang, Understanding Lustre File system Internals, OAK RIDGE NATIONAL LABORATORY, APRIL 2009.

12. John Bent, Garth Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, Meghan Wingate, PLFS: A Checkpoint File system for Parallel Applications, Supercomputing '09, Portland, November 2009.

13. Marc Eshel, Roger Haskin, Dean Hildebrand, Manoj Naik, Frank Schmuck, Panache: A Parallel File system Cache for Global File Access, 8th USENIX Conference on File and Storage Technologies (FAST '10), San Jose, February 2010.

14. Paul Nowoczynski, Jason Sommerfield, Jared Yanovich, J. Ray Scott, Zhihui Zhang, Michael Levine, The data supercell, Proceedings of the 1st Conference of the Extreme Science and Engineering Discovery Environment, Chicago, July 2012.

15. Torben Kling Petersen, Inside The Lustre File system, Seagate Technical Paper, 2014.

16. Jan Heichler, An introduction to BeeGFS, Available at http://www.beegfs.com, November 2014.

17. Junsup Song and Dongkun Shin, Performance Improvement with Zero Copy Technique on FUSE-based Consumer Devices, IEEE International Conference on Consumer Electronics (ICCE), January 2014.

18. George Neville-Neil, Too Big to Fail, Visibility leads to debuggability, ACM Queue, December 1, 2014.

19. Alexander Thomson, Daniel J. Abadi, CalvinFS: Consistent WAN Replication and Scalable Metadata Management for Distributed File systems, FAST'15 Proceedings of the 13th USENIX Conference on File and Storage Technologies, Santa Clara, February 2015.

20. Inside HPC Newsletter, Chameleon Testbed Blazes New Trails for Cloud HPC at TACC, Available at http://insidehpc.com/2016/10/uctacc-cloud-computing-testbed-helps-pittsburgh-supercomputing-center-get-ahead-of-the-game, October 2016.

21. Linux Filesystem in Userspace – FUSE. Available at https://github.com/libfuse/libfuse, February, 2017.

22. The GNU C Library (glibc). Available at https://www.gnu.org/software/libc/, February 2017.