Eventually Consistent Namespace in SLASH2

Pittsburgh Supercomputing Center

Last Update Date: 05/19/2010

The goals of SLASH2 are to support namespace and file data replication among cooperating metadata servers (MDS). To maximize availability, each MDS in SLASH2 should be able to operate independently. And an access to a *remote* source can potentially suffer in terms of performance, coherency, and priority. Administrative involvement is sometimes needed to make better use of SLASH2 because we don't use complicated distributed algorithms.

1. Overview

SLASH2 provides a global namespace to facilitate file sharing among multiple supercomputing sites such as the Pittsburgh Supercomputing Center (PSC). In such a heterogeneous environment, metadata servers are maintained by different administration groups and the network between them can have high latency and low bandwidth. We also cannot assume that any metadata server or network link is up and responsive all the time. Maintaining a single physical copy of the namespace shared by metadata servers in such an environment will at best result in suboptimal or even unreliable performance. The only way to go is for each metadata server to work on its own copy of the namespace. They of course should collectively represent a single logical namespace. But do not expect a fully consistent namespace like we would see in a cluster file system.

Fortunately, a global namespace does not have to be consistent all the time to be useful in practice as long as we exercise some caution when we design and use it. People are already learning to expect different things from remote and local accesses by the wide adoption of

distributed protocols like NFS. In SLASH2, there is no single authority on the entire namespace. But we do require that each directory in SLASH2 be managed by a single MDS. We call the MDS the owner of the directory. Or we can say the MDS owns the primary copy of the directory. This means that the ownership of the namespace is divided among all the MDSes in the system. There is no single point of failure in terms of namespace operation.

In SLASH2, we assume that only the owner of a directory has the write authority on the directory (this can change in the future if we implement dynamic ownership exchange). This makes sure that the contents of any directory are deterministic. In addition, the ownership of a file or a directory is determined by its creator. If a user creates a file or a directory, the file or directory is owned by the corresponding local MDS.

There are a couple of good things coming right out of this ownership arrangement: (1) Regardless of what is happening on other parts of the namespace, a MDS can be assured that the contents of its own directories are always up-to-date. (2) If only local users create files and directories under a locally owned directory, they constitute a subtree that is fully managed by the local MDS. If this MDS is partitioned away from other MDSes, its clients can continue to work within the subtree without any problem.

If all MDSes only worked in locally owned directories, then there would be no need for a global namespace. So the second big piece of the SLASH2 namespace puzzle is that each MDS must propagate the contents of its directories to its peers. This propagation is done asynchronous in the background. The upshot is that if no MDS updates the namespace for a while, all MDSes will eventually have identical copies (i.e., mirrors) of the namespace. This is called “eventual consistency”. It makes sure that any client in SLASH2, regardless of its location, will eventually see all the files in the entire system. Otherwise, only a divine god can do this.

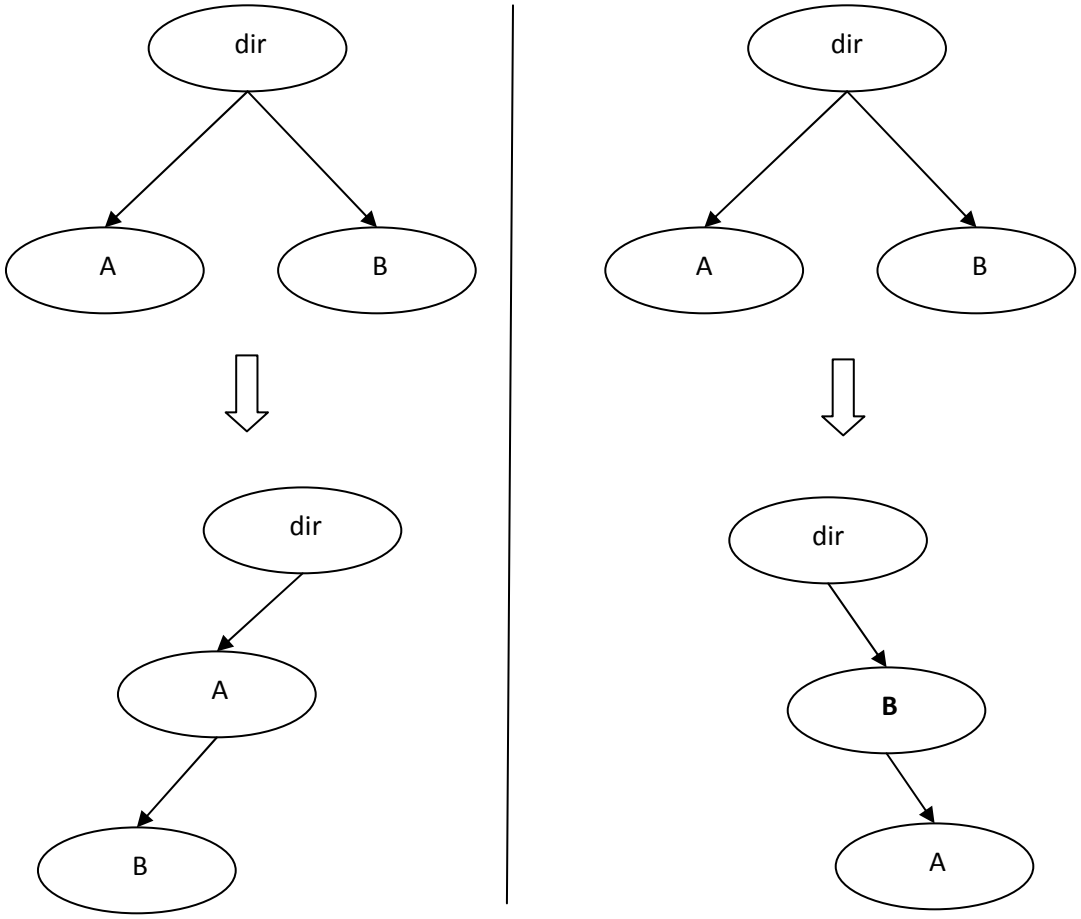
Building a global namespace in a wide area network is different from doing the same thing in a cluster environment. We have to make some concessions to achieve this:

(1) A directory write (e.g., mkdir) must go to its owner. If the owner is a remote MDS, it will take longer than when the owner is local. And if the MDS is down or partitioned away from the network, the write will be refused. If the MDS is busy, the write will be blocked.

(2) A directory read (e.g., readdir) is always done locally. This means we can return stale results if the directory is owned by a remote MDS. However, unless the relevant portion of the namespace is constantly changing at a fast pace, the result is probably a good one in practice. By the way, a user can find out the owner of a directory by examining its SLASH ID.

In other words, a SLASH2 client enjoys what we call “local performance” most of time except when it wants to create a file or a directory under a remote directory.

The above decision choices are made after considering usage patterns by potential SLASH2 users as well as the estimated amount of efforts needed to implement them. At one time, we considered creating a file or a directory *locally* even if we can’t contact the MDS that owns the parent directory (i.e., disconnected operation). That would achieve local performance even for directory writes at all the time. We even came up with some clever schemes (e.g., tagging *@site-name* after file names) to resolve merge conflicts. However, there is one big hurdle we could not overcome easily. As shown in the following diagram, if two MDSes change the namespace structure in a conflicting way by moving directories around, it is hard to resolve them (we can also limit these operations in a disconnected mode). The fact that a user can easily create a self-contained local subtree probably makes these efforts not worthwhile.



Site 1 Site 2

A third feature we want SLASH2 to have is called "early propagation" or "out-of-order propagation". Normally, a MDS would send out directory updates in the order of their occurrences. But sometimes, the local version of a directory can be way out-of-date with its owner's copy. However, if an operation is performed on a remote site, we would like to see its effect immediately on the local site that initiates the operation. Otherwise, there is no guarantee when the update will be propagated back and people might get confused.

Since SLASH2 provides a global namespace, each file and directory should be identified by a globally unique SLASH ID. A SLASH ID is a 64-bit quantity with two parts: 16-bit for the site ID and 48-bit as an uniquifier within a site. There are different ways to assign an uniquifier. One possibility is to use the ID decided by the local storage layer (e.g., ZFS)[1]. Another one is to use a random number generation whose period is 2^48 (Now I think about it, we can just use numbers from 3 – 2^48 sequentially, 2 is reserved for the root, to follow the old tradition). Either way, a SLASH ID is issued by the creator of a file or a directory. It allows a MDS to identify the owner of the corresponding file or directory.

2. Directory log

Each time a directory write happens, the owner of the directory has a responsibility to propagate the update to its peers so that their namespace copies converge. A MDS uses a directory log entry to store the update until it has been applied to all MDSes in the system. It is easy to justify that a directory log entry should have the following information:

- Update sequence number, 8 bytes
- Site ID, 2 byte
- Target site ID, 2 bytes
- Operation code (rmdir, unlink, create, link), 1 byte
- ~~Name of the parent directory (we don't really need it, names can be long).~~
- SLASH ID of the parent directory, 8 bytes,
- Type of the target, 1 byte
- Permission of the target, 1 byte
- SLASH ID of the target, 8 bytes
- Size of the entry, 2 bytes
- Checksum (including name), 8 bytes
- Name of the target, up to 256 bytes (only needed for a creation)

[1] The problem with this method is that we have to create a file in the underlying file system first before we can find out the ID assigned to the file.

Each directory update is assigned a unique update sequence number. Because a name can have different length, the size of a log entry can differ.

How do we store and manage these log entries? One way is to create per MDS log entry for each directory update. Once log entries are created, they are sent to other MDSes to apply them locally. These log entries should be stored on disk as long as they are needed. If a MDS is down, a log entry destined for it can stay for a long time. All updates can be stored in fix-sized data file like *update.site-id.000001*, *update.site-id.00002*, etc. We don't mix log entries sent to different MDSes in the same log file. Otherwise, a slow or down MDS can prevent us from recycling the log files.

The problem with this method is that if we have more than a couple of MDSes, the disk I/O and capacity used for logging will be increased substantially. For example, if we have 10 MDSes, then we would write 10 log entries for a single directory update, using ten times more storage. This is not a scalable solution.

But if we use one log entry for all the MDSes, we may need a way to track which MDSes have already applied a log entry. Storing log entries in some kind of indexing structure (e.g., B+tree) can be complicated. At this point, it seems a much simpler and coarse-grained recycling algorithm is the way to go. We recycle on a per log file basis, not on a per log entry basis.

As shown in the following diagram, log entries are stored sequentially and broken into separate log files. Each log file has a fixed number of log entries (their size can differ because each log entry can have a different size). We can put these log files under a special directory (say *.local* under the root), using file names like *update.1*, *update.2*, etc. To simplify, we don't track the availability of each log entry slot within a log file. This is because unlike a log used in a journaling file system, we can't guarantee that update log entries will retire in order, some log-receiving MDS can be slow or even down. Log generation and application are not guaranteed to be on the same pace. They happen on different MDSes.

0	1	……	4095
4096	4097	……	8191

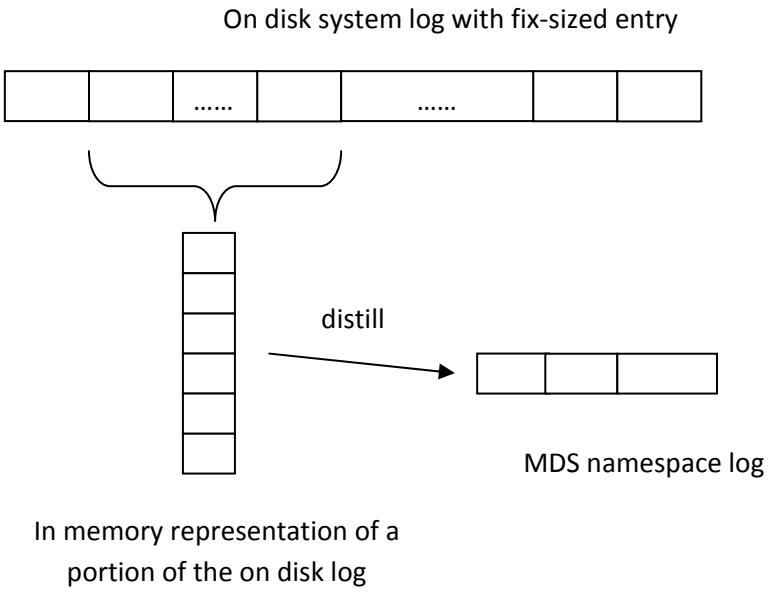
Log file format

Site ID	Outstanding Sequence Number
Site ID	Outstanding Sequence Number
Site ID	Outstanding Sequence Number
……	……
Site ID	Outstanding Sequence Number

Log application progress table

There is also a table used to track the progress of log application on all the MDSes. This table is written to disk periodically to save time in case of a crash. On the other hand, a MDS is expected to ignore a resent log entry and reply success with the highest log sequence number it has already applied to its namespace mirror. Obviously, the lowest sequence number of all MDSes can be used to retire log files.

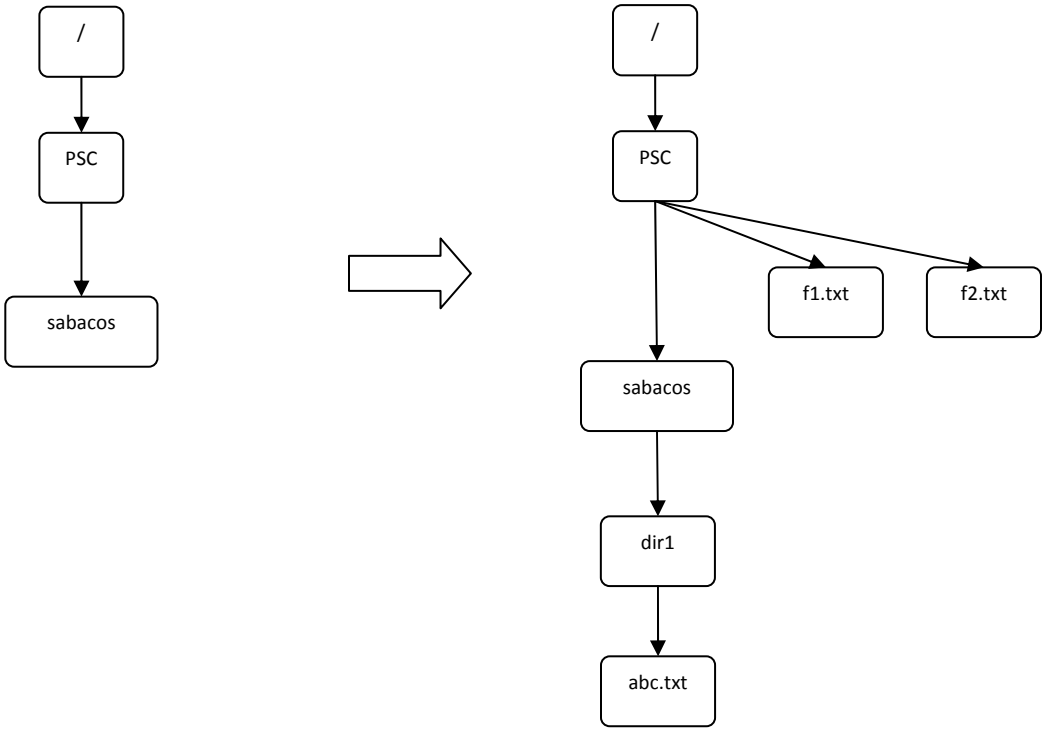
Whenever a MDS does a directory update, it performs the operation (in a local file system like ZFS), writes to its system log, and then replies to the client. The creation of directory log entries can be deferred because they have a different lifespan compared to the system log. It is also vary-sized. Currently, SLASH2 only logs for BMAP and CRC updates. How do we merge the existing log code base with the new directory log entry requirement is not quite decided at this point. One way is depicted in the following diagram.



Basically, we maintain a sliding window of the on-disk system log in memory and write them out to a separate namespace log file that contains directory update log entries distilled from the memory contents.

> If we don't hack user-space ZFS libraries, how does a MDS perform an operation in ZFS and log it properly before replying to the client? Normally, we have to make sure that any modified metadata by an operation won't be written before its log is written (i.e., write-ahead-logging used in a journaling file system). See section 7 for more discussions.

Once a MDS receives a bunch of log entries, in what order does it apply them? If we could treat each directory individually, then applying log entries would be conceptually trivial. However, directories are interdependent. Specifically, a child can't appear before its parent does and a parent can't die until all its children are removed. Let us look at an example:



The namespace transition depicted in the above diagram (from left to right) is achieved by the following steps:

(1) create PSC/f1.txt (local operation)
(2) mkdir sabacos/dir1 (remote operation, because sabacos is owned by another MDS)
(3) create dir1/abc.txt (local operation)
(4) create PSC/f2.txt (local operation)

If a third MDS is going to replay these changes, it must process (3) after (2). However, it should be able to perform (4) before (2) even if the former happens later. That way, a slow or down MDS that performed step (2) won't prevent *f2.txt* to show up under *PSC* in the third MDS.

How do we make sure that (3) is applied after (2)? We can use the idea proposed by the Leslie Lamport to establish a total ordering of namespace update events in the entire system. Specifically, we need to do the following:

(1) After a creation (i.e., mkdir) happens on a remote site, the update sequence number of the creation is passed back to the requesting MDS. The MDS guarantees that its future update sequence number will be larger than this one.

(2) To request a deletion (i.e., rmdir) on a remote site, the requesting MDS must pass its current update sequence number to the remote MDS. The remote MDS guarantees that it will use a higher update sequence number afterwards.

Note that only a remote directory update can introduce a dependency that must be reflected in the assignment of update sequence numbers. With this, the above four operations can be numbered as follows (assume that the local site ID is 5 and the remote site ID is 6):

(1) 0987,5: create PSC/f1.txt (local operation)
(2) 2224,6: mkdir sabacos/dir1 (remote operation)
(3) 2225,5: create dir1/abc.txt (local operation)
(4) 2226,5: create PSC/f2.txt (local operation)

Note that the site ID occupies the least significant bits of the update sequence number – it is used to break ties. Now if we strictly apply the log entries in the order of update sequence numbers and site ID, then (4) won't appear before (2). This means that the application of (2) serves as a barrier to all subsequent log entries, whether they are actually related or not. This is not an ideal situation. Note that using total ordering does seem to make it easier to support lock migration (we can easily order updates to the same directory coming from different MDSes).

Note that there is no such thing as ALL the log entries because they come and go all the time. A receiving MDS cannot do a topological sort of them. Worse, some of the log entries may never come.

So how about we ignore the logical order between log entries and just apply them when they arrive into some kind of backup data structure? When the logical order is satisfied, we can present them in the regular namespace that is visible to the clients. That way, we also don't have to store log entries in an on-disk indexing data structure so that we can apply them in some logical order.

It turns out that this bold move is feasible, but we need to make sure that a SLASH ID is never reused throughout the lifetime of SLASH2. We can use a generation number coupled with an inode number to achieve this. That way, we don't have to worry about a file or a directory is removed and then recreated using the same name. We can also use a random number generation number that won't repeat itself after a long period. Or just use numbers from 0 to 2^48 (assume we create 1,000,000 files in a second, 2^48 / 1000000 / 60 / 60 / 24 / 365 = 8 years). See Section 3 on how to apply log entries in details.

Two final thoughts:

(1) At least in the initial implementation, we should go for the simplest route to make sure we understand the mechanisms and there are no major holes in the design.

(2) We probably need to develop a tool to manually sync up a MDS's namespace with the global namespace. Any protocol, no matter how perfectly designed, can break down due to a software bug. See Section 6 for more details.

3. Access by SLASH ID

Normally, a SLASH client mounts on its local MDS and uses names to access files in the namespace. However, since a MDS needs to process a request forwarded by a remote MDS, it must be able to process a request based on a given SLASH ID.

From the perspective of a log-receiving MDS, such a SLASH ID (like an NFS handle used by an NFS server) can appear anywhere in the hierarchy, and an exhaustive search through the regular namespace is obviously too expensive to consider. Therefore, a MDS needs a way to locate any file or directory given its SLASH ID efficiently. Note that SLASH2 is a user-level file system.

One way to do this, similar to Zest, is to create hard links to files and directories in the regular namespace. The path to a hard link is entirely determined by the bits in a SLASH2 ID. By doing this, we can re-use existing namespace mechanism although we do have to violate the convention that a directory can't have hard links other than dot and dotdot. Collectively, these hard links constitute what we called "immutable namespace"[2]. The files in the regular namespace can be renamed or moved around, but their hard links constructed from the SLASH2 ID remain intact. This immutable namespace should remain inaccessible from the regular users by tweaking the permission bits.

The immutable namespace acts like a convenient back door to the regular namespace. It does come with a price – adding a hidden link to every new file and directory (file creation is slowed by a factor of two). Incidentally, on an I/O server, SLASH2 uses the immutable name space exclusively to store file data accessible (i.e., readable and writable) only by their IDs.

Another way to achieve this is to use our own directory format, be it B+tree or some hash format. Currently, SLASH2 uses native directory format in a MDS. In other words, a mkdir in SLASH2 is translated to a mkdir on the underlying file system. However, it is possible to use a

[2] The upper layers of the immutable namespace are created when we format the file system with a utility. They never change (i.e., immutable). The immutable namespace is also called by-id namespace, versus the regular by-name namespace visible to SLASH2 users.

customized directory file that appears a native regular file to the underlying file system (e.g., ZFS).

If we go this route, we don't have to maintain two physical namespaces on each MDS (one regular and one immutable). We still use bits in the SLASH ID to format a hierarchy to find the file or directory. A special ID can be used to identify the root. Perhaps the root IDs can be the same on all MDS (except for the site ID part), unlike IDs of other files or directories. And the ownership of the root is shared by all MDSes. To maintain consistency at the root directory level, each MDS automatically creates subdirectories that name all the sites in the system on its own when they register with each other. These subdirectories are owned by their respective MDSes. We can still allow updates in the root, but they won't be propagated to other MDSes[3].

The immutable namespace essentially provides a hidden namespace that allows us to access any file or directory by its SLASH ID instead of its name. With this capability, we can apply log entries as they arrive regardless of whether their interdependencies have been satisfied or not.

(1) If a log entry that creates file arrives before its parent directory is created, we just create the container for the parent directory in the immutable namespace. It is indexed by the parent SLASH ID, which we know. Note at this point, we don't know the name and attributes of the parent yet. When the log entry for making the parent does come later, we can then fill in the right name and attributes. And this is when the parent directory becomes part of the user visible namespace.

(2) If a log entry that deletes a directory arrives before all its children are removed, it is removed from the real namespace first. Later on, when a log entry to remove a child comes in, it is removed from the immutable namespace only.

    Removing a link in the namespace is essentially removing a mapping from a name to SLASH ID in the parent directory. We might need to add a children count to a directory, so that we can recycle it when all its children have been removed.

When we apply log entries, we locate the target by a SLASH ID. The target will show up or disappear from the real namespace as a side effect. Removing files and directories from the immutable namespace is not required for correctness; it is done to save disk space and better performance.

With the help of the immutable namespace, the interdependencies between directories are removed. However, we can still have an ordering issue within the same directory. Suppose we

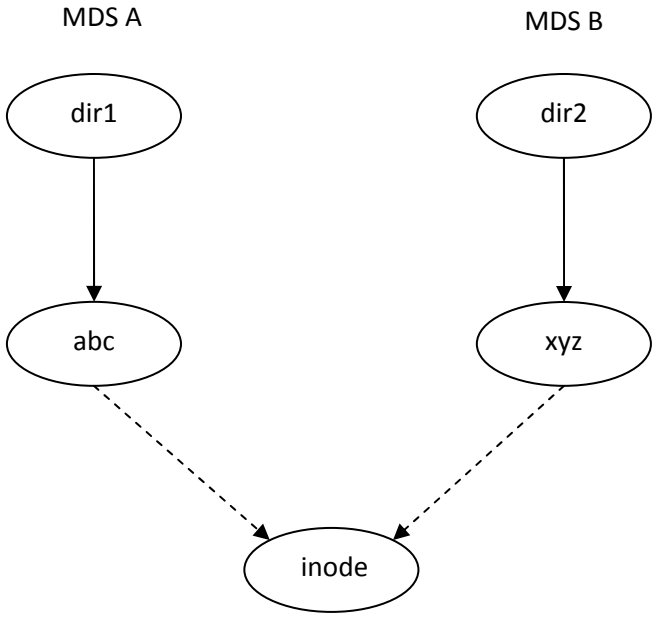
[3] Alternatively, we can designate that the root is owned by MDS #0. Then MDS #0 must be up all the same. This can reduce the availability of the system.

add file abc.txt and then remove it, generating two log entries for an MDS. How can we ensure that the two log entries are applied in that order? It seems to me that we can solve this problem by maintaining per-directory update sequence numbers (now that we don't need them to order among directories).

What about a link across different directories? In the following diagram, dir1 and dir2 are owned by two different MDSes: A and B respectively. We perform the following steps in time order:

(1) Create dir1/abc on MDS A
(2) Create a link dir2/xyz to dir1/abc on MDS B
(3) Remove dir1/abc on MDS A

Do we have to apply the log entries generated by the above three operations in some kind of "correct" order? I think no, as long as we apply the log entries on MDS B in order.



If we remove dir2/xyz on MDS B later, then on a third MDS, we can see dir2/xyz for some time before it is finally removed although it does not really exist anymore in the first place. This is because we apply ALL log entries one at a time (i.e., we replay the history in full). Only a directory owner has an authoritative view of its contents all the time.

4. Namespace operations

Now let us run through a bunch of scenarios to see if the above assumptions hold. Go from simple ones to complex ones. The hard part is to convince ourselves that we have thought of everything.

4.1 Lookup

Initially, we started from the idea that all lookups proceed locally based solely on the copy of the namespace on the local MDS. If the parent is owned by the local MDS, the result is definitely authoritative. Otherwise, the result is not 100% dependable. However, unless the relevant portion of the namespace is constantly changing at fast space, the result is probably a good one in practice. Excising extra patience on the part of users can further improve the chances of the result being good.

If we want to track how far away a MDS's directory is from its authoritative copy, we need to maintain a generation number for each directory. When a MDS creates or deletes a file in a directory owned by a remote MDS, it will know if its copy of the directory is stale or not. If its copy is stale, it can forward all lookups to the owning MDS.

But there is a problem here. If the network to the owning MDS is down, we will be blocked on a directory read (we are already blocked on write if the network is down) even though the local copy is probably good enough.

Yet another option is to only return those entries in a directory that owned by the local MDS.

Right now, we settle on the idea to always perform lookups locally.

4.2 Create/mkdir

To create a file or a directory, the request must be sent to the MDS that owns the parent directory. There are two cases here:

(1) The creation can be done locally. In this case, the MDS performs the operation and schedule the directory updates to be sent to all peer MDSes.

(2) The creation can't be done locally because a remote MDS owns the parent directory.

For the second case, we have two choices:

(1) The local MDS can forward the request to the remote MDS, which performs or rejects the creation locally. The method is simple, but slow if you want to create lots of files under a directory owned by a remote MDS.

(2) Take the ownership of the parent directory temporarily (i.e., lock) and create locally. To do so, we have to sync up the local version of the directory first. Granting and returning ownership is not a trivial thing. In addition, we can't predict the behavior of a user. Unless a user creates/deletes lots of file in a directory, the cost of transferring ownership won't be amortized. Another snag is that applying logs can be affected by dependencies among directories (this hurdle can be overcome by using the immutable namespace).

Of course, if the remote MDS that owns the parent directory is down, then a directory write will be stalled. There is no (easy) way to around it.

So we prefer (1) to (2) if we have to create a file or directory under a parent that is owned by a remote MDS. Luckily, to guarantee local performance, a user can easily create a personal directory and then work under it. Then the directory is owned by the local MDS regardless of who owns the parent. Afterwards, all the namespace operations underneath the directory will be local and authoritative.

Note that it is possible that the parent directory has been removed and the local MDS has not been notified about the removal (it is not the owner of the parent). If so, a creation will fail and a user may be confused because its local namespace copy seems to allow the operation. We can use some implementation trick to avoid this (not sure how difficult it would be).

4.3 Link

It should be the same as create above. If we create and remove a link to the same target several times within the same directory, we must apply these updates on a remote MDS in the right order. This means it is hard to piggyback results. Note that the link itself does not have a SLASH ID.

What if we associate with each update with an update sequence number (no need for site ID here because this is within a directory)? If there is a conflict on a name between an incoming log entry and the current contents of the directory, we just let whoever has a higher number to win. If we go this route, we may need to use a customized directory format – each directory entry must now contain an update sequence number. With the mechanism, we can apply a log entry any time we want. And we can piggyback a log entry on the reply message to the link request.

4.4 Unlink/rmdir

An unlink operation also modifies the namespace. It should be handled by the MDS that owns the parent. This is done similar to the create case. There are three cases here:

(1) If the target does not exist in the local copy of the namespace, return ENOENT immediately.
(2) Send the request to the MDS that owns the parent of the target. If the local MDS owns the directory, perform the operation locally and return.
(3) Forward the request to the remote MDS that owns the parent of the target and perform the operation there.

The fact that the target exists locally means that the log entry used to create the target has already arrived at the local MDS. Note that if the same file gets deleted and recreated in between, it will have a different SLASH ID. If we match a log entry and a target by the SLASH ID, not by name, we should be fine. So we should be able to piggyback the unlink/rmdir result back to the requesting MDS without worrying about the application order of log entries.

4.5 Rename

A rename can involve as many as *two* different parents; each of them can in turn owned by different MDs. A rename will be broken down as a copy followed by an unlink, so it could be tricky. Note that a rename can be done over an existing target. This means there could be three separate actions: remove the existing name, create the new target, and remove the old name. Unless we want to implement a distributed transaction, this is the easiest way to go.

Finally, if an operation is performed remotely, we would like to witness its effect locally by the time our request returns. If piggybacking result is difficult to implement, another way is to synchronize directory contents on-demand (i.e., apply all the pending log entries). However, to achieve that, we need to be able to locate all the log entries for a directory efficiently. Right now, we plan to store and process all log entries for all directories sequentially in time. See Section 7 for more details.

5. Namespace protocol

Namespace protocol is used to request a directory update remotely and to propagate a directory update to other MDSes. There is no RPC to do a namespace lookup.

5.1 Create (create and link)

A MDS can request a remote MDS to create an object on behalf of its clients. On reply, the remote MDS can optionally tell the requesting MDS to apply the change locally. The requesting MDS must give the two SLASH IDs: one for the parent and one for the target to be created. So we need the SLASH ID before the target is created.

5.2 Remove (rmdir and unlink)

This is similar to the create case.

5.3 Apply log

It is used by a MDS to propagate directory updates to other MDSes. We expect each log entry corresponds to a single directory update. We need to store the current update sequence number with each directory.

Two remaining issues:

(1) What kind of credentials do we use for authentication? A simple Unix (uid, gid) pair or Kerberos?

(2) We might need a reply cache similar to the one used in NFS. But that is an optimization.

6. Add and remove a MDS

This section discusses the need to add and remove a MDS *permanently*. The directory log protocol should be able to deal with temporary availability issues such as a down MDS or a network link.

Because of the existence of a hidden namespace that is indexed by SLASH ID and the fact that a SLASH ID is never reused, we do not need to construct or remove files and directories in an order so that they appear or disappear in the regular namespace naturally (i.e., a child is dependent on the existence of its parent). This essentially removes the interdependencies among directories and files. Therefore, an incoming MDS can simply contact all the MDSes, asking them to send the contents of directories to itself.

Each MDS in turn scans its namespace, looking for directories owned by itself. For each of these directories, it sends the current contents of the directory to the new MDS, and future updates as well. There could be a performance issue for a MDS to find all the directories owned by itself among others in the big namespace. The time when the new MDS have a complete copy of the namespace is determined by its peers (how cooperative and fast they are).

It is easy to stop a MDS from propagating its directory updates to an existing MDS, so removing a MDS from the picture should be easier than adding a MDS.

If need be, we can ask system administrators to cooperate to make sure that we add or remove one MDS at a time. However, since we don't rely on any complicated quorum based

distributed protocol, it probably does not matter if we add or remove more than one server at a time.

Instead of rebuilding a new MDS from scratch, another way is to take a snapshot of the namespace copy maintained by a currently running MDS and use that as a baseline to bootstrap the new MDS faster. Specifically, we need to do the following (this is not an automatic process, administration involvement is needed):

(1) Quiesce a currently running MDS. It should stop initiating new directory updates or accept log entry updates from its peers. Lookups (i.e., directory read) should be allowed. Although if each individual object can re-synchronize with its authoritative copy on its own, we can miss or add some directories in the process if we don't quiesce the MDS.

    By refraining from replying to any log application requests, the log entries won't be purged from the log files on its peers. They are needed for the new MDS.

(2) Make a copy of the namespace and transfer it to the new MDS.

(3) On the new MDS, register with all MDSes of its intention to join the group, including the site ID of the MDS where the new MDS gets a copy of the name space.

(4) After all MDSes has responded, the new MDS is in. Now, the MDS whose namespace image we just copied from can be upgraded to be fully functional.

Note that we can add a new MDS even if some of the network is down – simply adding the new MDS's information into the tables of existing MDSes will do it. We should also be able to abort the above process at any point. This can happen if the new MDS fails to contact some MDSes. We should provide a tool for a MDS to remove one of its peers (this can be as simple as to remove an entry from the log application progress table).

Note that we don't attempt to implement a quorum-based distributed protocol among the MDSes. That would be very difficult and probably not really needed. We expect each MDS to have on-site or on-call 24x7 supporting staff.

7. Customized directory format

7.1 Early log entry application

Sometimes a user needs to create or remove a file in a directory that is owned by a remote MDS. Even if the local MDS does not have a complete view of the directory (i.e., some log

entries have not applied yet), the user would like to see the effect of the operation immediately on the local MDS. Note that we should create a log entry for the requesting MDS anyway in case the reply to the request gets lost (should be use TCP instead of UDP? should we do heartbeat exchange?).

One way to achieve "early propagation" or out-of-order application of a log entry is to attach each name in a directory with an update sequence number. That way, if a log entry that refers to the same name arrives, we can decide whether to accept or reject the action based on the update sequence number it carries. The contents of a directory look like the following (they consist of a sequence of directory entries):

Update sequence number	Name	SLASH ID	Flags
456	abc.txt	xxxxxx	xxxxxx
768	123.dat	xxxxxx	xxxxxx
……	……	……	……

Even if a file is removed later, we should keep an entry in the directory (but mark it as deleted), so that we won't apply any earlier log entry that creates the same file. This entry can be removed as soon as the lowest applied update sequence number of the directory is no less than its update sequence number. We don't need to use per directory update sequence number if we make sure that a MDS never sends out a second batch of log entries until its previous batch has been acknowledged. This may be not true, because the first batch of log entries can be lost in the traffic (if not using TCP).

For example, a local client creates file *abc.txt* and the reply carries back update sequence number 456. If a log entry comes in later that wants to remove *abc.txt*, it will be rejected if the log entry has a lower update sequence number.

The per-entry update sequence number is there to protect out-of-order (i.e., early) log entry application. It will become useless after it is smaller than the update sequence number for the directory itself. We could do some tricks to reclaim its disk space, but it is really a very minor issue. Not many people complain about disk capacity these days.

Even though the disk space is not our concern, this design does force us to use our own directory format.

On a second thought later, I realize that we can use the SLASH ID to ignore an older log entry or accept a newer log entry that refers to the same name. This is because we never reuse a SLASH ID – each new file, regardless of its name, is assigned a new unused SLASH ID. This way, we won't need to add a SLASH ID field to each entry in a directory. We still need to save an unlinked entry in the directory, using the flag field to mask it off. When the corresponding log entry comes in, the space taken by the unlinked entry can then be removed. We also don't need to store a per-directory update sequence number either. Similar argument also applies to section 7.2 below.

Furthermore, if a client wants to remove a file say 123.txt, it must see it locally first. This means that the log entry corresponding to the creation of 123.txt has already been applied. However, it is possible that 123.txt has since then been removed and recreated. And the local MDS doesn't know about it.

7.2 Log before reply

The above discusses the need for a customized directory format from the perspective of applying logs. The following discusses the need for an MDS to honor its promise for clients.

Right now the namespace of an MDS is implemented directly with a tree stored in the underlying file system (e.g., ZFS). For example, if there a directory named PSC in the namespace of the MDS, there is a corresponding directory with the same name PSC in the local file system. This makes it hard to do true write-ahead-logging (WAL) required by an MDS unless we hack the underlying file system.

In the case of ZFS, we have no control when a namespace update (e.g., mkdir) will reach the disk. ZFS does use a no-overwrite strategy to maintain self-consistent in metadata structures. But that does not guarantee a namespace update has been made permanent when the call to perform the operation returns. Making everything synchronous will slow down performance.

Fortunately, there is a way to work around this issue while still using a third party file system like ZFS that we don't need to hack into. First, we no longer maintain an identical hierarchy of the MDS namespace in a local file system. Instead, we use only our immutable namespace that is indexed by SLASH ID. All namespace information is recorded in the regular files in this immutable namespace. The real namespace can of course be derived from the immutable namespace.

Second, we associate each update of the namespace with an update sequence number. This helps us to identify which log entries have already applied and resolve any potential conflicts.

With above considerations, we can do the following on an MDS:

(1) Write a log entry (can be written into a file that always does synchronous write) to describe our intention first.
(2) Perform the operation in the underlying file system (e.g., ZFS).
(3) Send a reply to the client.

By the time the client receives a RPC reply to its request, the operation is guaranteed to have at least been logged on disk. The underlying file system is free to cache the effect of the operation as necessary. Although we don't know when the underlying file system actually commits the operation on-disk, we can just wait reasonably enough time (say 30 seconds) to make sure that the corresponding log entry is no longer needed. We can even force a sync if the log space becomes low.

7.3 Implementation Issue

There are two ways to implement a customized directory format. One way is to build a tree inside a *regular* POSIX file. The problems are (1) we don't have access to disk space allocation in the user space, so holes in the file still occupy disk space (we can truncate a file, but cannot punch a hole in a file); (2) more importantly, we can't control the order of disk writes in the user space. To maintain the integrity of a B+tree, we may need to update several B+tree blocks in a single directory update.

The other way is to hack ZFS code. Regardless of the internal data structures of a ZFS directory, they guarantee its consistency after a crash (*we probably still need to modify ZFS logging code to take the new fields in a directory entry into account, but the infrastructure is already there*). We only need to check if a certain update is missing in the directory or not after a crash. If we modify ZFS code, we must store SLASH2 directories as ZFS directories, but with augmented entries. We should be able to use immutable name space only.

8. Remote Access

If we can't read and write a remote file, it is not very useful even if we know that the file exists. However, maintaining a fully coherent file system across different sites is hard to achieve. We can have two different approaches to achieve remote access.

Currently, SLASH2 supports replication within one site. The smallest unit of replication is a block in a file (default is 128MB). Each file in SLASH2 has a block map to track the locations and states of its replicas on a per-block basis. Each block is further divided into slivers (default is 1MB) and there is a CRC associated with each sliver. Whenever a block is written, its other replicas are invalidated. We need to either do garbage collection on I/O servers or the MDS has to maintain the replicas to bring them up-to-date.

By the way, garbage collection is a tricky issue. To improve performance, we want to do it asynchronously. But we also need to avoid reading stale data. The edgy cases include O_TRUNC open and truncate to zero. Note that from the coherent point of view, a truncation means a write between the new EOF and the old EOF.

8.1 Support replication across sites

The main goal of this approach is to support data locality. If there is no local replica, then a remote access will be forwarded to the owning MDS. And a remote access is treated equally as a local access.

The hard part of using replicas across sites is how to resolve potential conflicts. One idea is to fork and let each MDS modify its own fork. When reading a block map of a file, the local copy is always favored if it exists. If a MDS has a full replica of a file, we can also implement a detach-on-write strategy: a file becomes independent (and therefore renamed) as soon as it is written. Alternatively, we can let a user pick and choose file contents on a per block map basis.

There are two ways to support data fork. One way is to use plain POSIX files. There is a directory for each SLASH2 file, and under the directory there is a file for each fork. Another way to implement fork is to use openat(O_XATTR). The idea is to use extended attributes for each fork.

8.2 No replication across sites

This is a simpler solution. All accesses to a file must be done by its owning MDS and replication is done only within a site. To make things even simpler, we give any remote access a lower priority in two senses: (1) it can't introduce any conflict; (2) it won't get notified if a conflict happens. Specifically, this means the following:

A remote access is allowed if there are no other existing accesses that conflict with it. For example, if a file is being read, the remote access cannot be a write. If another remote access wants to access the same file in a conflicting manner, it will be rejected. If a local access wants to do the same thing, it will be granted immediately. The remote access will be rejected immediately without notification.

These ground rules make the file coherent without implementing any complicated distributed algorithms. All accesses are not equal. After all, you don't own a remote file and you should probably talk to whoever owns it first. SLASH2 won't solve every problem for you.

Within the same MDS, however, SLASH2 still supports fully coherent data access. For example, if more than one client is writing into the same block map, client-side caching will be turned off.

9. Reference

A reference could be *Distributed Operating Systems & Algorithms*, by Randy Chow & Theodore Johnson. The review of this book is not good, but I haven't found a better book on this topic yet.

10. Afterthoughts

02/12/2010

Paul Nowoczynski and I realized when discussing implementation details that we actually need to maintain two namespaces no matter what. One is indexed by names; the other is indexed by SLASH ID. A link in the namespace is created when we add a (name, id) to its parent. So we need at least two links to a file or a directory. A file or a directory is always searchable by its ID, but not necessarily by its name.

The immutable namespace we mentioned in this document is really a namespace indexed by IDs. Since the namespace is built on ZFS, the namespace needs both ZFS inode number and SLASH ID. When we descend from the root of the namespace, ZFS inode numbers are used. At the leaf level, SLASH IDs are used to reconstruct a pathname in ZFS to find a child of a SLASH directory.

05/10/2010

In a namespace update operation, we log the namespace operation *before* the operation is actually performed. Whether the operation is going to fail or succeed, we don't know at this point. What happens if there is a crash afterwards?

Fortunately, ZFS guarantees self consistency after a crash. By comparing the log contents with what's in ZFS, we can figure out if the operation has been performed or not. If not, we will attempt to do it. If the operation was a failure, it does not hurt to try and fail again.

05/19/2010

Continue my thoughts started on 05/10/2010.

After spending some time studying ZFS source code and reading blogs about ZFS, I am convinced that we can log a namespace update *before* ZFS commits the corresponding transaction or the associated intent log, but *after* we know the operation is a success (never log anything in vain). Also, in our log entry, we can record the ZFS transaction group number that can tell us deterministically whether we need to replay our log entry. This way, we can tie

SLASH2 log, which is external to ZFS, with ZFS transaction. Note that we replay our journal after ZFS replays its logs.