Data Migration in a Distributed File Service

Mallik Mahalingam, Christos Karamanolis, Lisa Liu,
Dan Muntz, Zheng Zhang
Computer Systems and Technology Laboratory
HP Laboratories Palo Alto
HPL-2001-128
May 23rd , 2001*

E-mail: {mmallik, Christos, lisaliu1, dmuntz, zzhang}@hpl.hp.com

data migration, distributed file system, Storage Area Networks (SAN), storage management

As distributed systems span the globe, placing data near the point where the data are accessed is becoming important to improve client performance and to reduce network load. With the advent of the utility model in storage services, it is necessary for storage service companies to provide quality of service guarantees to meet customer needs. Adaptive data migration will be required by storage service providers to guarantee these service-level agreements. Storage customers will require the ability to transparently migrate data among different storage service providers based on, for example, pricing differences or QoS issues.

Data migration is also useful for reducing resource contention (thus increasing scalability) by breaking up "hotspots" in the file system. Data migration plays a significant role in increasing the scalability and performance of distributed file systems.

In this paper, we examine mechanisms for transparent, flexible data migration in the context of our experimental distributed file service, DiFFS.

Data Migration in a Distributed File Service

Mallik Mahalingam, Christos Karamanolis, Lisa Liu, Dan Muntz, Zheng Zhang,

Hewlett-Packard Labs
1501 Page Mill Rd, Palo Alto, CA 94304, USA
{mmallik,Christos,lisaliu1,dmuntz,zzhang}@hpl.hp.com

Abstract

As distributed systems span the globe, placing data near the point where the data are accessed is becoming important to improve client performance and to reduce network load. With the advent of the *utility model in storage services*, it is necessary for storage service companies to provide quality of service guarantees to meet customer needs. Adaptive data migration will be required by storage service providers to guarantee these service-level agreements. Storage customers will require the ability to transparently migrate data among different storage service providers based on, for example, pricing differences or QoS issues.

Data migration is also useful for reducing resource contention (thus increasing scalability) by breaking up “hot-spots” in the file system. Data migration plays a significant role in increasing the scalability and performance of distributed file systems

In this paper, we examine mechanisms for transparent, flexible data migration in the context of our experimental distributed file service, DiFFS.

1 Introduction

There is a significant change taking place in the resource model of Internet-connected entities. Resources such as computing, network and storage on the Internet are being provided in much the same way as electricity and gas have been provided in the past—via a “utility” model. A number of industry [1] and academic [2] research projects are trying to provide a “Utility-Based“ infrastructure to make these resources available automatically, “on demand,” and to guarantee requested service levels. Utility service providers, such as storage service providers [3-5], guarantee certain levels of service based on the resource usage model for their resources.[3]

One challenge facing Internet Data Centers (IDCs) is how to provide efficient and cost-effective storage services that satisfy the evolving requirements of Application Service Providers (ASPs). Service Providers and Data Centers must handle fluctuating and unpredictable workloads. These workload variations can cause parts of the storage infrastructure to be over-stressed while other parts may be under-utilized. Migration can increase performance of storage systems in these environments by addressing issues, including:

1. Decrease the distance between data and clients accessing the data.
2. Reduce contention by using migration to break up “hot-spots.”
3. Place data at a cost-effective location.
4. Allow for dynamic expansion (or reduction) of the data set.
5. Provide non-disruptive access to critical data.

There are many factors that may contribute to poor performance, including insufficient server processing power, storage contention, network congestion, and poor data placement. Migration is also used to respond to resource changes. For example, in a utility-based infrastructure, processing power, network bandwidth and storage can change dynamically. Efficient data migration is essential in this model.

Several data migration solutions have been proposed by both academia and industry. The NFSv4 [6] standard proposes mechanisms for revalidating NFS file handles in the case of file system migration, but it does not introduce any protocols for the actual data migration and it does not consider migration at a finer granularity. The Andrew File System (AFS) [7] provides mechanisms for logical volume migration. Migration is made transparent to the application through the user of a volume location database (VLDB)—a global data structure (maintained consistently across the system) that maps logical volume identities to physical locations (servers). Both NFS and AFS allow migration at a very coarse granularity, either an entire volume, or file system. They do not support migration at the granularity of individual files or file sets.

The following are three examples where file-level migration is desirable: 1) to move the personal files of users close to the point of use; 2) to move certain files away from “hot spots;” 3) move the files to a file system better suited to serve their content type. GASS[8] is a system that supports a range of migration mechanisms. However, it requires explicit support from the applications (a special API), which makes this solution impractical for many *applications*.

This report addresses the data migration issue from a generic perspective. It proposes mechanisms for both volume and object-level data migration. The problem is investigated for widely distributed storage services that follow the "file system" abstraction. The main requirements for the proposed mechanisms in this context are:

- Scalability: they must scale for the size of current and future data sets.
- Granularity: they must support data sets that vary anywhere from a small set of files (perhaps a single file) to logical volumes.
- Transparency: they must be non-intrusive to the applications that are deployed above the storage service.

The decision about what part of the data set to move, the location to which it is moved, and the timing of the migration are policy issues, which will be addressed in future work. This paper focuses on mechanisms required for migration.

The proposed mechanisms are designed in the context of DiFFS, a distributed file service architecture currently under investigation in HP Labs. An overview of DiFFS is given in section 2, followed by a discussion on "location transparency" for data in that architecture. Volume-level migration is discussed in section 4 and object-level migration in section 5. Section 6 addresses issues related sharing objects that are migrated. Related work is discussed in section 7 and the paper concludes in section 8.

2 DiFFS: a scalable distributed file service

DiFFS is a distributed file service that uses a partitioning approach to address the problem of contention for storage resources [9]. It is designed to tolerate host and communication failures without sacrificing performance for failure-free operation [10]. DiFFS can be deployed above multiple types of physical file systems; objects can be placed in a file system that is best suited to their type and size. DiFFS achieves scalability by partitioning the storage and controlling shared access on a per-partition basis. The basic building block of the DiFFS architecture is a partition server. Each partition server in the system has exclusive control over a set of logical volumes (Figure 1). Every logical volume is associated with an identifier (LV-ID) that is unique in the system.

A directory in a file system is a list of entries, each one containing a reference to an object (file or directory) in the system. Traditionally, a directory entry includes the inode number (inode#) and name of the referenced object. However, files and directories within DiFFS may reside in any partition, thus requiring *cross-partition references*. The inode# field in the directory entry is augmented with the ID of the logical volume where the referenced object resides. The latter requires extensions to the directory entry structure. To provide these extensions in a generic way, DiFFS directories are implemented as normal (data) files in the underlying physical file systems. This approach has two advantages: 1) DiFFS can be deployed on top of any physical file system; 2) directories can be implemented in various ways that are optimal for different access and search patterns.

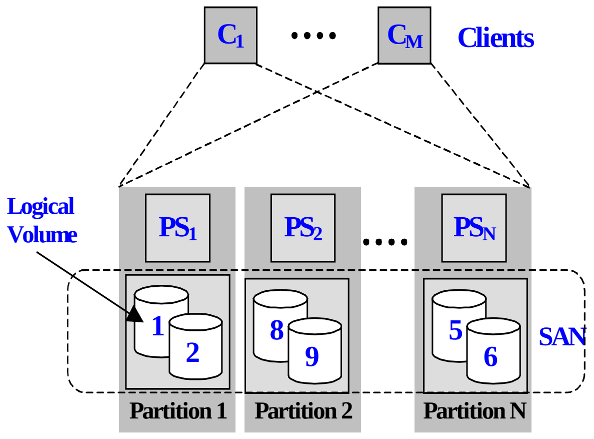

Figure 1: DiFFS Architecture

The location of where the objects reside is transparent to the clients. Object location is determined by some "placement policy" at creation time.

3 Location transparency in data access

The DiFFS architecture provides a location-independent representation of objects, which is similar to that of AFS. Typically, when a client needs to access a file or a directory, a lookup operation is performed first. The result of the lookup is a file handle that is used by the client to perform future accesses. The role of a file handle is to provide a way to identify the object at the server end and is opaque to the client. The file handles are often cached at the clients for performance reasons.

A "lookup" operation in DiFFS is performed as depicted in Figure 2. The execution is described in a way that is independent of the access protocol [9] used. For example, a lookup for file "/etc/passwd" is performed in three stages: (1) retrieve the file handle for the root inode—this information is typically stored on every partition, in this case it is retrieved from partition 1; (2) read the contents of directory "/" and construct the file handle for directory "etc"; (3) read the contents of "etc" and construct the file handle for file "passwd".

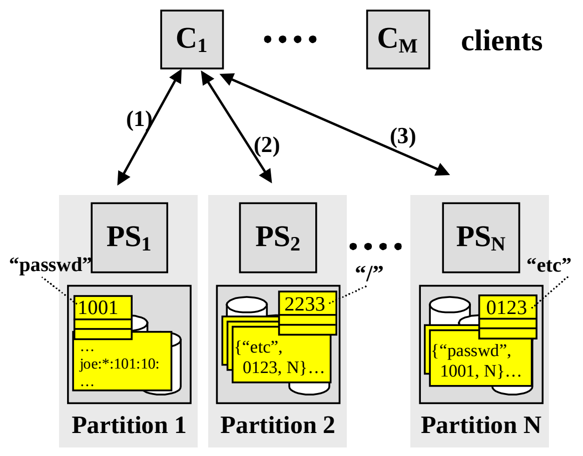

Figure 2: Execution of lookup ("/etc/passwd") in DiFFS.

3.1 File Handle

The structure of a DiFFS file handle is shown in Figure 3.

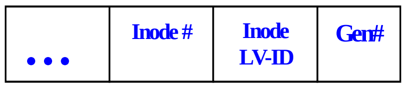

Figure 3: Structure of a file handle in DiFFS

A file handle is composed of the following information that is kept within the directory entry of the object.

- Inode # of the object
- Logical Volume ID (LV-ID). The purpose of the LV-ID is to identify:
    - The Logical volume where the object resides.
    - The partition server that is acting as a "custodian" of the logical volume that contains the object
- Generation # of the inode.

DiFFS provides a way to represent objects in a location-independent manner by embedding the LV-ID in the directory entry. The actual custodian of the logical volume is retrieved as described in the following paragraphs.

3.2 Mapping Logical Volumes to Physical Partitions

In order to access an object, a client has to retrieve the actual custodian of the logical volume referenced in the file handle of the object. This mapping information has to be maintained in a consistent way in the system and it must be accessible by the clients in an efficient way.

This section introduces two methods for maintaining consistent system-wide mappings of logical volumes (LV-IDs) to partitions.

3.2.1 Global mapping state

A global mapping table is maintained in the system, similar to the volume location database (VLDB) in AFS. The mapping table may be maintained and kept consistent by a Storage Management Service. It can be replicated or cached at various locations for performance and availability. A mapping service running at the clients provides mapping resolution by querying the table (either a replica of the table or some locally cached part of it). The mapping table is read-mostly. It is updated only in the presence of volume migration (see section 4). Volume migration is not a frequent event in the system. Therefore, a protocol that guarantees strict replica consistency [11-13] can be employed for updating the global mapping table.

3.2.2 Distributed algorithm

Each partition server maintains an *ownership table* with the LV-IDs of all the volumes the server is responsible for (owns). The table is assigned a version number; every time its contents change, the version number is incremented. Upon delivery of an access request for an object, the table is consulted to check whether the local server hosts the corresponding volume or not.

A partition server finds out about the ownership tables of other partitions, as a side effect of cross-partition transactions in the system. The ownership tables of other partitions are locally cached on the partition and form its *neighbourhood* information. The cached tables are maintained loosely synchronized with their master copies on the owner partition servers. The local copies of *neighbours'* tables are consulted when, for example, the partition server receives a request for an object that does not reside in a local volume. Upon lookup, in addition to the file handle, the client is also provided with a *hint* for the partition server it should contact for that object. This information is cached on the client and is used when the object is accessed. It is just a hint about the location of the corresponding volume, because the volume may not reside in the partition indicated by the lookup. Such scenarios can occur when volumes are migrated across partitions. Volume migration is discussed in detail in section 4.

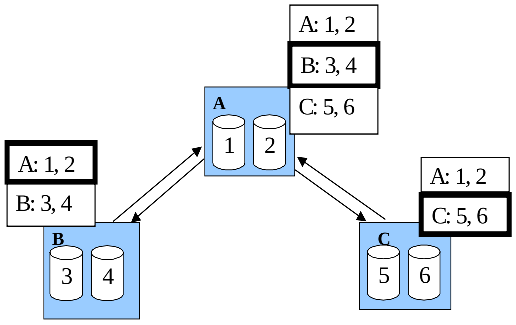

Figure 4: Initial Mapping of LV-IDs and Partition Ids.

Figure 4 shows an example of three partitions, each containing two volumes. Each partition's own table is depicted with a thick borderline. Outgoing arrows indicate "knowledge" of another partition's ownership table (the neighbourhood of that partition).

As a result of migrating a volume from partition A to partition D (volume 1, in the case of Figure 5), both A and D have the up-to-date ownership information about themselves, but partitions B and C have now stale tables about A (depicted as shaded boxes in the figure).

Stale maps are updated in one of two ways. First, any cross-partition transaction (for example, general DiFFS traffic) is piggybacked with the version number of the ownership table of the originating partition. Thus, partition servers can find out whether the locally cached copies of others' tables are out-of-date. In the latter case, the up-to-date tables are explicitly requested from the corresponding partitions.

Second, the update of stale tables can be initiated when a partition receives a request for an object that resides in a migrated volume. For example, consider the configuration of Figure 5. A file "foo" resides in volume 1, which has just migrated from partition A to partition D. A parent directory of "foo", named "dir", resides in volume 3/partition B. A client does a lookup for "foo" in "dir" and gets back a file handle, which indicates that "foo" is located in volume 1/ partition A (B consults its out-of-date cached table about A). When a following request from the client arrives at partition A, it is forwarded to partition D (or the client is asked to re-send the request to D), the new location for volume 1. Note, that A has an up-to-date table for itself and D. In this case, the payload of the client request is required to carry information about the location of the parent directory of the object. That is, directory "dir" in volume 3/partition B. Using this information, partition server A can infer that partition B has out-of-date information about the location of volume 1. The updated tables for A and D are explicitly forwarded to partition B.

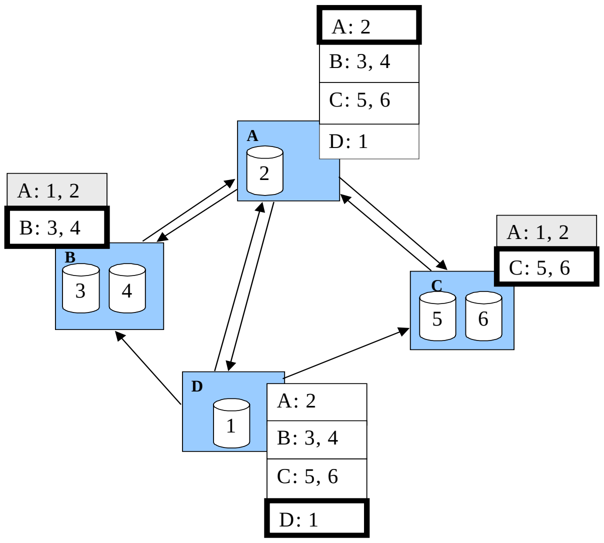

Figure 5: Mapping of LV-IDs and Partition Ids after Logical Volume "1" migration.

The algorithm presented here guarantees that the information about volume to partition allocations is maintained in the system in a distributed way. Consistency of this distributed state is maintained by means of a lazy technique. The algorithm has the advantage that eliminates any single point of failure (or performance bottleneck) in the system, but it is clearly more complex than the approach described in the previous section.

4 Volume-level migration

DiFFS supports volume migration as a fundamental mechanism to ensure the robustness of the architecture. Moving a volume is usually an expensive operation, the cost of which is proportional to the size of the volume. DiFFS builds the distributed file service on top of multiple volumes and therefore can afford to use small volumes to aid incremental, more fine-grained migration.

There are two basic flavors of volume migration in DiFFS: changing the ownership of a logical volume and physically relocating the logical volume to a different device. A combination of these two is used in some cases.

The following examples demonstrate scenarios where volume-level migration may be required:

1. When a partition server is retired, all the volumes it owns need to be reassigned. This requires change of ownerships for the volumes.
2. Likewise, when a partition server is overloaded, the ownership of some of its logical volumes needs to change to some other partition server that is idle or less loaded.

3. When a device that hosts multiple logical volumes becomes a “hot-spot” (or the network path that leads to the device is highly congested), then one or more logical volumes need to be migrated to different device(s).

If volume migration across devices is required, then it is performed first. There are many well-known online volume migration techniques; a process using the mirroring technique of the LVM package available in Linux is as follows:

1. Form a logical volume with the mirroring option by creating a mirror set that contains the target device (all writes go through the replica of the logical volume located on the target device “secondary” and reads go through the primary logical volume).
2. When mirroring is complete, a replica of the logical volume is created
3. Mirroring is cancelled and the primary of the mirror set is freed up but note that the logical volume ID is not changed.

If the device on which the logical volumes reside becomes a “hot-spot,” the above procedure is sufficient to address the problem.

4.1 Protocol for changing the custodian of a logical volume

When reassigning the ownership of the logical volume, say, LV-ID 1, from partition server Ps to partition server Pd (with or without device migration), the following steps are performed:

1. Freeze I/O (new requests are put into a pending state) to the LV-ID 1 at Ps.
2. Dismount the file system from the logical volume LV-ID 1 at Ps. As a side effect, outstanding I/Os for the LV-ID 1 buffered in Ps’ cache are flushed.
3. Send a request to Ps to mount the volume.
4. Upon receiving an acknowledgement from Pd, update the mapping to reflect the change of ownership of the volume. We have discussed two alternatives in section 3.2.

4.2 Failure Analysis

4.2.1 Failure Model

We assume the following failure model:

Hosts fail by crashing; they do not exhibit malicious (Byzantine) behavior.

Messages may be not sent or not delivered due to host crashes. Also, messages may be lost due to network partitioning. On recovery from any such failure, the communication session between two hosts is re-established. Messages delivered during the same communication session between two hosts are always delivered in order. This condition is guaranteed by using TCP as the communication protocol.

Consistency of the local object-store is guaranteed, despite failures. This property is ensured by mechanisms of the physical file system, such as journaling [14], soft updates [15] or recovery procedures (fsck) [16].

Log entries are written synchronously and atomically.

4.2.2 Failure analysis on volume-level migration

In order to guarantee fault-tolerance, we propose that the partition server Ps, from which the volume is migrating, initiates a recovery protocol after step (3) (described in section 4.1). The recovery protocol is initiated when Ps does not receive an acknowledgement from partition server Pd. Ps may not receive an acknowledgement from Pd for various reasons:

a) The new partition server (Pd) is not reachable due to host or network failures
b) The new partition (Pd) is still available but the mount process is still in progress
c) The new partition server (Pd) received the request, mounted the file system for the logical volume but the acknowledgement was lost due to network failure.

For all of the above failure cases, the recovery protocol at Ps remounts the file system for the logical volume and releases all pending I/Os to proceed normally. Request for updating mapping of logical volume to the partition server Pd is not sent, thus, leaving only the original partition server Ps access the logical volume. There is a small possibility for having both partition servers mounting a file system for the same logical volume. However, Pd is instructed to dismount the file system from the logical volume when the connection with Pd is re-established.

5 Object-level migration

In some situations, it is desirable to move individual objects instead of the whole volumes. For example, the files that constitute the personal profile of a nomadic user need to be migrated close to the physical location where they are accessed from. What objects (individual or working-set) should be migrated, when and where they should be moved to are policy issues that are handed by SMS. Once SMS identifies the object(s) that need to be migrated, migration mechanisms move those objects to the specified locations.

The requirement here is to move individual objects (files or directories) across volumes that potentially belong to different partitions. Aggregated object-level migration can be used for achieving "working-set" or "group" migration.

5.1 Protocol for object level migration

The Protocol for "online" object-level migration is composed of the following 5 phases:

1. Create a consistent replica of the object in the new location. Any requests to access the object are put on hold while creating the replica. A relaxed requirement will be to allow reads to proceed.
2. Create forwarding information at the original location
3. Release pending requests and perform a transparent file handle revalidation
4. Update the namespace references to reflect the new location of the object
5. Garbage-collect the forwarding information when all the affected references in the namespace have been updated and all the cached file handles are revalidated.

However, if "offline" migration is used, only phases 1 and 4 are required, to create the object replica and update the namespace references to point to the new location. "Offline" object migration is more applicable to Hierarchical Storage Management (HSM) environments that target optimization of data storage rather than in environments that target performance.

For example, consider that partition servers P1, P2, P3 and P4 own LV-IDs 1,2,3,4 respectively. SMS initiates migration of object "/a/x" (inode #1001) that resides on LV-ID 2.

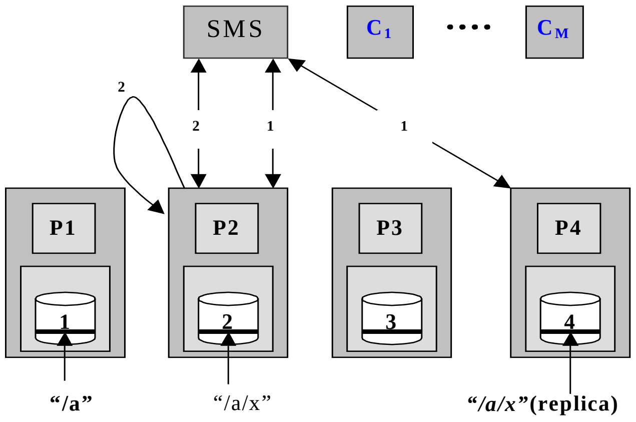

Figure 6: Execution of object-level migration for object "/a/x"

In ***Phase 1***, a replica (Inode# 1010) is created in LV-ID 4 for object "/a/x" (Figure 6). The replica can be created using either user-level primitives such as "cp" or using kernel operations. The actual replica creation can be performed either by SMS or by the partition server that owns the logical volume on which the object resides. While creating the replica, new requests that modify the object are put on hold.

A request is sent to P2 in ***Phase 2***, for creating a forwarding information for object "/a/x" <Inode #1001, LV_ID 2> to reference the new location <Inode #1010, LV-ID 4> (Figure 6).

In ***Phase3***, all pending requests for accessing object "/a/x" are returned back to the clients with a reply message that contains a status that "object has moved" along with the new location information. Upon receiving the reply message, clients update the cached file handle. Subsequent access to the object would land at the new location <Inode #1010, LV-ID 4> (Figure 7). The mechanism also revalidates stale cached file handles in the clients, which point to the original location of the object <Inode# 1010, LV-ID 2>.

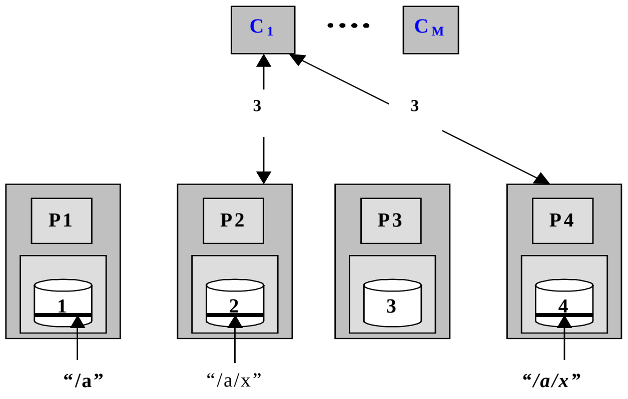

Figure 7: File-handle re-validation process

5.2 Phase 4: Updating namespace references

Once an object is migrated, it is necessary to have the namespace references to the object point to the new location. Namespace updates can be accomplished either aggressively or lazily. Each approach entails different tradeoffs and design complexities.

5.2.1 Aggressive Update

Aggressive namespace update is performed immediately after creating a consistent replica of the object. It is implemented by performing "***re-link***" [10] operations on all the namespace references that point to the migrated object.

For example, "/a1/x1" and "/a2/x2", could point to the same object, say, <inode# 1001, LV-ID 2> (*hardlinks*) (Figure 8a). The back pointer references [10] kept for object <inode# 1001> at LV-ID 2 identify the parents of this object. Assume that "/a1" (Inode# 1900) resides in LV-ID 1 and "/a2" (Inode# 1800) resides in LV-ID 3**.** Partition servers P1, P2, P3 and P4 own logical volumes LV-ID 1, LV-ID 2, LV-ID 3 and LV-ID 4 respectively.

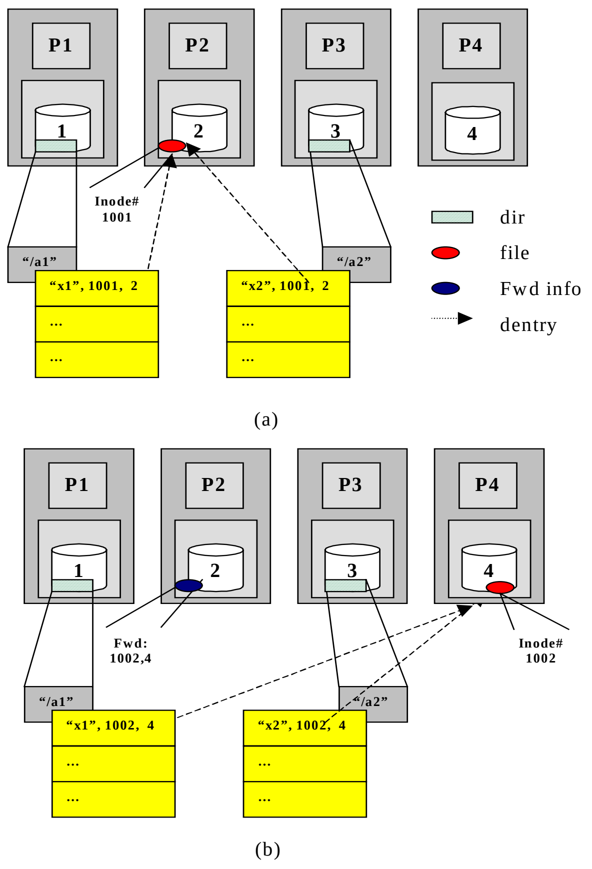

Figure 8: Aggressive namespace update on "/a1/x1 and "/a2/x2", when object (Inode #1001) referred by "x1" and "x2" migrates at LV-ID 2 migrates to LV-ID 4 (Inode# 1002)

When object <Inode# 1001, LV-ID 2> migrates, to say, <Inode# 1002, LV-ID 4>, we can traverse the back pointer references kept for this object and locate the namespace references from "/a1" <Inode #1900, LV-ID 1> and "/a2" <Inode #1800,LV-ID 3>. Update requests are sent to partition servers P1 and P2 to update the two references (Figure 8b).

5.2.2 Non-Aggressive update

Non-aggressive update is initiated when a request for access to the migrated object lands on a partition server where forwarding information is maintained. That is, a server that owned a logical volume where the object *used* to reside. This server updates the namespace reference by decoding the object and parent directory object information from the file handle that is used while accessing the object. In order to support the latter the file handle has been extended to contain parent directory information as shown in Figure 9.

• • •	P-Dir Inode#	P-Dir LV-ID	Inode #	Inode LV-ID	Gen#

Figure 9: DiFFS file handle that contains parent directory information of an object for performing non-aggressive namespace update

For example, partition servers P1, P2 and P4 own logical volumes LV-ID 1, LV-ID 2 and LV-ID 4 respectively (Figure 10a) and object "/a1/x1" (Inode#1001) has migrated from LV-ID 2 to LV-ID 4 (Inode #1002). .

When a request for accessing this object arrives at P2, the following operations are performed:

1. P2 sends a request to P1 (where the parent directory object "/a1" resides) to change the directory entry for object "x1" to point to the new location <Inode# 1002, LV-ID 4>.
2. When P1 receives a request to update a directory entry, it acquires a write lease on the directory file that is being updated, invalidates cached copies of the directory file at the clients and updates the directory entry. P1 sends a message back to P2 with the status of the update operation indicating success or failure. If the corresponding directory entry has already been updated, P1 responds with "success".
3. Upon receiving a "success" message, P2 performs an unlink operation locally to drop down the reference count.

The result of non-aggressive namespace update of object "/a1/x1" is show in Figure 10b.

The case of objects with multiple references (*hardlinks*) in the namespace is handled in the same way as described above. A reference is updated when it is used (through a lookup operation) to generate a file handle to access the original position of the object. However, there is no guarantee that a reference to an object is always eventually used to access it. Thus, the non-aggressive approach cannot put any upper bound to the time required for all references of an object to be updated.

A "lazy" namespace update process can be initiated to update all the namespace references for the migrated object by walking through the global namespace (in the background of normal operation). However, going through an entire hierarchy of global namespace to find the references would not be practical in real systems.

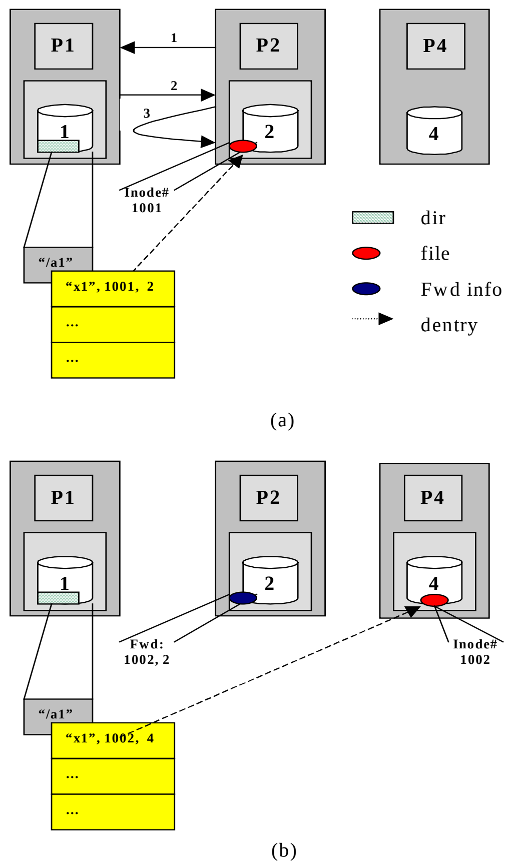

Figure 10: Non-Aggressive namespace update on "/a1" when object "/a1/x1" migrates from LV-ID 2 (Inode# 1001) to LV-ID 4 (Inode# 1002).

When the parent directory of an object migrates prior to migration of the object itself, the parent directory information that is contained in the file handle of the object may be stale and thereby making the update to the namespace references difficult. This can be handled using one of the following methods:

A. When a directory migrates, messages are sent to the partition servers for each of its children. Each message contains the original location and the new location of the migrating directory. This information is retained at the partition servers, and is consulted under the following conditions:

    a. When a file handle revalidation request (see section 5.3) for a child of the directory lands at the partition server.

    b. When a child migrates from the partition server, and the namespace update process is initiated.

B. When a revalidation request of a file handle for an object arrives at the partition server, both the object portion of the file handle and the parent directory portion (P-Dir Inode# and P-Dir LV-ID in Figure 9) of the file handle are revalidated.

5.3 Phase 5: Garbage collection of forwarding information

In order to provide an efficient garbage collection process, clients are required to perform *periodic* revalidation of the file handles they cache. That is, a client has to revalidate every cached file handle at least every "Tr" units of time. In this way, a partition server that maintains forwarding information for a migrated object is guaranteed that after "Tr" time there are no more file handles cached by clients that point to the old location of the object (condition 1).

The passage of time Tr is *not* a sufficient condition to garbage-collect the forwarding information for the corresponding object. Another necessary condition is that there are no more references in the namespace pointing to the original location of the object (condition 2). This condition is met by keeping track of the reference (link) count of the original object location. In the case of *aggressive* updates, the latter condition is met in bounded time. However, in the case of non-aggressive updates, there is no upper bound for the time required to maintain the forwarding information for a migrated object.

The above two conditions are necessary and sufficient to garbage collect "file" objects. However, they are not sufficient in the case of migrated directories. A third condition must be met in that case: the back-pointers of the children of that directory have also to be updated to point to the new location of the directory before the forwarding information is discarded (condition 3). To achieve this, the reference (link) count of directory objects is extended to reflect not only references from the namespace (from parent directories) but also references from children objects (back-pointers). The forwarding information for a migrated directory can be garbage collected when all the above conditions (1, 2 and 3) are satisfied.

5.4 Failure Analysis

In order to provide fault tolerance, an entry is created in an "intention log" at each step of the migration process. Entries are reclaimed when the process is done. The intention log is used for performing "undo" and "redo" operations in the case of failures.

5.4.1 Failure analysis of object-level migration that uses aggressive namespace update.

In the following paragraphs, we outline the possible failure cases and corresponding recovery operations for the migration process.

Create Log

Begin: Create copy of the object (1)

Done: Create copy of the object (1)

Begin: Create Forwarding Info (2)

Done: Create Forwarding Info (2)

Begin: "re-link" (4)

Done: "re-link" (4)

Reclaim Log

Figure 11: Object-level migration using aggressive namespace update process with "Intention Log".

A. Step (1) fails but the original copy of the object is still intact and no damage has been done to namespace that has reference to original object. The failure could leave the copy process partially complete and the process could have failed due to network partitioning or host failures.

"Undo" the changes by removing the incomplete copy that migration process might have created.

B. Step (1) was successful but there has been failure before going to step (2) due to host failure

Either the copy that migration process has created can be removed or move forward to step (2) and perform "redo" as the original copy is still in tact since update to the namespace has not been done.

C. Step (1) was successful, but a request was sent out to create forwarding information (step (2)) and the acknowledgment was lost due to network partition or host failures.

The original copy is probably inconsistent but the duplicate copy that migration process has created is available. Perform step (2) over and over until we get ACK from the partition server that contains the old copy of the object.

D. Step (2) was successful and before going to step (4) (*re-link process*), network partition or host failures occurred

The original copy of the object is lost and it contains forwarding information. New copy of the object is available, so redo step (4) until it runs to completion.

E. Step (2) was successful and failure resulted some where in the middle of step (4)

Perform step (4) until its completion. Step (4) is repeated for all the namespace that the migrated object has been referred from.

F. When reference count for the old object reaches zero (all the namespace references have been updated), forwarding information is kept for "Tr" time and the entry is freed up when it expires.

G. Reclaim the entries from the intention log.

5.4.2 Failure analysis on non-aggressive namespace update approach.

Failure analysis on the object-level migration that uses non-aggressive update is little different from aggressive namespace update approach since the namespace update process is decoupled from migration process. However both approaches have common phases, phase 1 and phase 2, therefore the recovery process described in section 5.4.1 for these two phases are still applicable here. We propose using the same mechanism "intention log" that we used in the case of aggressive namespace updates, to perform recovery process. Failure scenarios for the operations described in section 5.2.2 are described below.

A. Request is sent out in step (1) and due to host failure or network partitioning, acknowledgement was lost.

B. If acknowledgement was sent out in step (2) and the message was lost due network partitioning or host failures

C. Step (2) was successful but before going to step (3) there has been a host failure

D. Step (2) was successful but host failure occurred while performing step (3)

For all failure cases, the recovery process performs step (1) through (3), asynchronously, until it runs to completion. Since initiation and completion of namespace update process is logged in the "intention log", recovery process can be rescheduled even when partition server that executes the recovery process crashes.

5.5 Comparison of aggressive and non-aggressive namespace update approaches

The table below summarizes the characteristics of the two approaches for namespace updates described in the previous sections.

	Aggressive	**Non-Aggressive**
Data Structures	Requires additional data structures such as back pointers	No additional data structures
Life of forwarding Information	"Tr"	No upper bound
Migration of parent object	Need to aggressively update all the children	Requires extended file handle revalidation mechanism such as: Revalidating the object portion of the file handle as well as the parent directory portion of the file handle **Or** Propagating forwarding information to all the children when parent directory migrates and keeping the forwarding information of the parent directory for "Tr" time.
Namespace Updates	All namespace references are updated immediately after migration	No theoretical upper bound on how long the stale references live. A lazy update process can be employed, but may not be practical

6 Issues related to sharing objects

Typically, clients cache the contents of file system objects for performance reasons. The validity of the cached data and the duration that the cached data remains valid are controlled via mechanisms such as leasing. When an object migrates, the migration process first acquires a write-lease for the object and then performs the migration process. This is done to ensure that no modifications to the contents of the object are made while the migration is in progress.

When objects migrate, we not only have coherency related issues, but also issues pertaining to hard state of the objects such as locks. Hard state is migrated to the new custodian of the object. The migration mechanism proposed here handles the coherency related issues in the same way that it handles objects. When a client gets a message back from the partition server indicating the object has migrated, it updates location information and obtains a fresh lease from the new location.

7 Related work

Existing solutions do not address the level of granularity or the robustness and simplicity for performing efficient data migration supported by DiFFS. Implementations, such as NFSv4[6], address migration at file system level, which is insufficient for handling "hot-spots" at fine granularity. AFS proposes migration at the volume level. An AFS volume is a collection of a logical set of files such as a user's data. Migration of an AFS volume is performed by cloning the volume and then updating the VLDB. AFS volume-level migration is not suitable for environments such as mobile computing, where migration of a small "logical" set of objects is often required.

GASS provides data movement using *file cache* to applications using specialized calls on Unix systems. In order to take advantage of the supported mechanisms, applications must explicitly access remote files using specialized function calls such as *globus_gass_open (), globaus_gass_close (), etc*.

Object level approaches, such as that of OceanStore[2], provide migration mechanisms through traditional (implementing their own physical file system) and non-traditional interfaces (APIs) by utilizing location-independent object naming. However, DiFFS provides better migration mechanisms and higher granularity. Partitioning based approaches such as Slice[17] and Archipelago[18], make data migration difficult because the object location is determined at creation time at the clients using hashing functions; moving an object around involves propagating a new hash value to all the clients that access the object. This is an expensive process.

Other solutions such as Akamai FreeFlow [19] provide mechanisms for delivering contents efficiently, but it is primarily designed for web applications. It is not generally suitable for applications that Application Service Providers (ASPs) deploy.

Traditional solutions used in Hierarchical Storage Management (HSM) systems, such as DataThink[20], are focused on optimizing the storage space by moving inactive data from primary storage to tertiary storage media (e.g., tape libraries). These systems are not concerned with efficient data access.

Logical volume based migration approaches such as Veritas [21], support data movement but not migration of the control aspect of the data.

8 Conclusions

In this report, we have described two main mechanisms for data migration that address performance and scalability in a distributed file service which provides storage services across multiple geographic sites. The mechanisms described are simple, robust and designed to guarantee non-disruptive access to data when the system is performing data migration. We have outlined two different namespace update approaches for object level migration and compared them to stress the strengths of the aggressive approach.

The mechanisms proposed also handle soft state such as leases. Migration of hard state information (e.g., locks) will be addressed in the future.

DiFFS provides efficient data movement mechanisms and a variety of granularities in order to help IDCs address current and future demand for storage services. We have provided some examples and scenarios that are potential candidates where these data migration mechanisms can be used. Data migration mechanisms proposed in this report, help IDCs realize the true benefit of infrastructures based on a utility model.

References

1. Wilkes, J., et al. *Eos - The Dawn Of The Resource Economy*. in *HotOS-VIII Workshop*. 2001. Schloss Elmau, Germany.
2. Kubiatowicz, J., et al. *OceanStore: An Architecture for Global-Scale Persistent Storage*. in *ASPLOS 2000*. 2000. MA, USA: ACM.
3. StorageNetworks, http://www.storagenetworks.com.
4. StorageWay, http://www.storageway.com/.
5. ScaleEight, http://www.scale8.com/.
6. Shepler, S., et al., *NFS version 4 Protocol*. 2000.
7. Howard, J., et al., *Scale and Performance in a Distributed File System*. ACM Transactions on Computer Systems, 1988. **6**(1): p. 51-81.
8. Bester, J., et al. *GASS: A Data Movement and Access Service for Wide Area Computing Systems*. in *Sixth Workshop on I/O in Parallel and Distributed Systems*. 1999. Atlanta, GA, USA.
9. Karamanolis, C., et al., *DiFFS: a Scalable Distributed File System*. 2001, HP Laboratories: Palo Alto.
10. Zhang, Z., et al., *Cross-Partition Protocols in a Distributed File Service*. 2001, HP Laboratories: Palo Alto.
11. Schneider, F.B., *Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial*. ACM Computing Surveys, December 1990. **22**(4).
12. Karamanolis, C. and J. Magee. *Configurable Highly Available Distributed Services*. in *14th IEEE Symposium on Reliable Distributed Systems*. September 1995. Bad Neuenhar, Germany.
13. Mishra, S., L.L. Peterson, and R.D. Schlichting, *Implementing Fault-Tolerant Replicated Objects Using Psync*, in *Proceedings of the Eighth Symposium on Reliable Distributed Systems*. October 1989: Seatle, Washington. p. 42--52.
14. Preslan, K., et al. *Implementing Journaling in a Linux Shared Disk File System*. in *8th NASA Goddard Conference on Mass Storage Systems and Technologies*. 2000.
15. Gagner, G. and Y. Patt. *Metadata Update Performance in file Systems*. in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 1994.
16. Bovet, D.P. and M. Cesati, *Understanding the Linux Kernel*. 1st ed. 2001: O'Reilly.
17. Anderson, D., J. Chase, and A. Vadhat. *Interposed Request Routing for Scalable Network Storage*. in *Usenix OSDI*. 2000. San Diego, CA, USA: USENIX.
18. Ji, M., et al. *Archipelago: An Island-Based File System for Highly Available and Scalable Internet Services*. in *4th USENIX Windows Systems Symposium*. 2000.
19. Akamai, http://www.akamai.com.
20. DataThinK, http://www.datathink.com.
21. VERITAS, *Veritas Volume Replicator*, http://www.veritas.com/us/products/volumereplicator/.