Simplifying Administration Through Dynamic Reconfiguration in a Cooperative Cluster Storage System

Renaud Lachaize*†
Sardes Project
Imag-LSR and Inria
renaud.lachaize@inrialpes.fr

Jørgen S. Hansen
Department of Computer Science
University of Copenhagen
Denmark
cyller@diku.dk

Abstract

Cluster Storage Systems where storage devices are distributed across a large number of nodes are able to reduce the I/O bottleneck problems present in most centralized storage systems. However, such distributed storage devices are hard to manage efficiently. In this paper, we examine the use of explicit, component-based (command and data) paths between hosts and disks as a vehicle for performing non-disruptive storage system reconfiguration. We describe the mechanisms necessary to perform reconfigurations and show how they can be used to handle two management tasks: migration between network technologies and rebuilding a disk in a mirror. Our approach is validated through initial performance measurements of these two tasks using a prototype implementation. The results show that on-line reconfiguration is possible at a modest cost.

1. Introduction

Over the past few years, cooperative storage systems for clusters, i.e., storage systems using disks attached to regular nodes in the cluster, have been the subject of much attention, both at the file [14, 2, 5, 22, 11] and block levels [3, 21, 17, 7, 8]. The main interests of this approach is the reduction of the performance bottleneck found in centralized servers and local I/O buses. At the block-level, an additional advantage lies in the avoidance of expensive Storage Area Network (SAN) hardware, thanks to the use of the existing, off-the-shelf resources provided by the involved nodes. The cost-effectiveness of such "virtual SANs" has gained much attention since a significant share of clustered applications take advantage of the features (e.g. failover, strict locking semantics...) provided by intermediate layers[1] that rely on shared storage devices.

The downside of cooperative storage systems is that their management is largely unsupported, thereby making it difficult for them to provide the reliability and performance required by most high-end I/O intensive applications. In this paper, we investigate how the injection of active code into the data access paths between the clients and the disks of such a storage system can be used to perform management tasks in a non-disruptive and bandwidth efficient manner. Our objective is not to introduce yet another fault-tolerance algorithm (nor data redundancy scheme) but to provide a framework that allows administrators and developers to respond, without much effort, to specific needs regarding dynamic storage administration.

In cooperative storage systems for clusters, frequent intervention is necessary to ensure high reliability and good performance. Examples of such tasks are the replacement of a failed disk followed by the rebuild of redundant data or the management of the storage pool to increase its capacity and improve the I/O load balancing. These tasks are time-consuming and error prone for human administrators. In a cluster context, the lack of tools for distributed, software-based infrastructures providing storage services makes the administration even more complex and costly. In our approach, rebuilding a failed disk can be done by inserting the rebuild code at a point in the data access path where all updates are visible, i.e., on the same node as the data, thereby ensuring a bandwidth efficient rebuild process. This ability is also useful for other administrative tasks such as performance debugging.

High availability is also a crucial factor. While a disruption can be acceptable for some high performance computing applications relying on checkpoints, it is hardly the case for data servers. Thus, there is a strong need for clustered

*Institut National Polytechnique de Grenoble
†Contact Address: Projet Sardes, Inria Rhone-Alpes, 655 Avenue de l'Europe, Montbonnot, 38334 St Ismier Cedex, France

[1] Typical examples for such intermediate layers are cluster file systems [27, 30] and parallel database management systems [28].

storage systems that support non-disruptive and assisted reconfiguration. We adopt an approach where the actual management task is executed concurrently with regular client data manipulation—the only disruption is the actual injection of the code to handle the management task.

We are implementing these concepts within a component-based framework called Proboscis [13] that focuses on easing the construction of distributed storage services operating at a low level (block or object-based storage). The storage services are built from a set of code modules with functionalities such as striping or disk access scheduling resulting in a distributed data structure that makes the data access paths in the cluster concrete. Our storage management approach is based on modifying this distributed data structure at runtime. Our current prototype implementation can be used as a replacement for a SAN in cluster file systems using a block-based storage abstraction but can be extended to support object-based storage as well. Regarding on-line reconfiguration, the prototype implements the code injection necessary for the reconfiguration system. These mechanisms provide the necessary functionality for building higher level services such as data migration, snapshot or backup.

The rest of this paper is organized as follows. Section 2 provides a brief overview of the Proboscis architecture. The generic reconfiguration mechanisms provided by our framework are exposed in section 3. We describe the implementation of two detailed reconfiguration scenarios in section 4. Section 5 explores the context of reconfigurable storage systems. Finally, we conclude in Section 6.

2. The Proboscis Framework

Proboscis [13] provides a framework for sharing disks in a cluster of workstations. Its goal is to allow nodes in the cluster to share their local disks with low overhead thereby enabling cluster-wide storage systems to be based on these local disks. The current implementation supports block-based access to remote devices[2], and can be used as storage system for cluster file systems such as GFS [27] or GPFS [30]. Proboscis can alternatively be described as an extension of the works on network block devices [4], with a focus on modularity and adaptability. In this section, we introduce the components of the Proboscis framework that are used by the reconfiguration mechanisms and refer to [13] for a full description.

In the Proboscis framework, the data and control flow between storage devices and storage clients passes through a connection consisting of a series of *extensions* (code modules). These extensions implement tasks such as local disk access, network transport, RAID, disk sharing or storage client access to the framework.

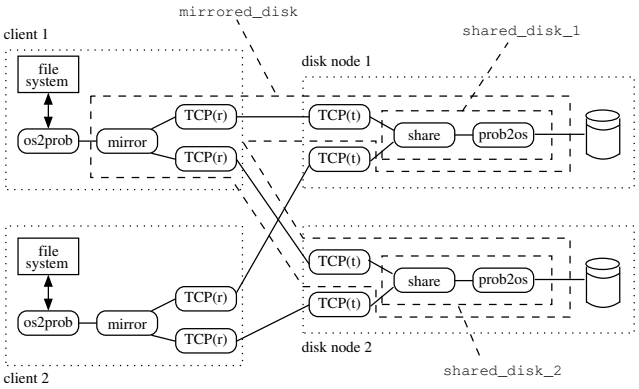

Figure 1. Example showing two clients using mirrored remote disks using proboscises

In the following, we will use the example in Figure 1 to describe the internal processing of the Proboscis framework. The example consists of two nodes "client 1" and "client 2" accessing a mirror of two remote disks on nodes "disk node 1" and "disk node 2". The two nodes exporting disks use the extension `prob2os` as an interface between the proboscis and the operating system block device layer. The disk sharing extension `share` is connected to `prob2os` to allow multiple clients to use the disk. An extension has two sides; one towards the storage device (the *tip direction*) and one towards the storage client (the *root direction*). On each side there are a number of *sockets* that can be connected to other extensions sockets on the opposite side using *joints*, e.g. `share` has a single socket in the tip direction but multiple sockets in the root direction.

A series of extensions can be published as a named proboscis, allowing it to be used as a building block for other proboscises. In our case, the two disk nodes publish the proboscises `shared_disk_1` and `shared_disk_2`, and the mirror extensions `mirror` can connect to these instead of specifying a series of extensions. A name service (see section 3.2.2) will resolve a request for a named proboscis and prepend an appropriate network extension to the named proboscis if necessary. In the example, the remote disks are accessed across an Ethernet network, resulting in the `TCP` extensions being interposed between e.g. `mirror` and `shared_disk_1`.

A network extension actually consists of two parts: a root side part (e.g., `TCP(r)`) and a tip side part (e.g., `TCP(t)`), but to the other extensions it acts as a local extension. The network communication itself is not visible to

[2]Our framework has nonetheless no strong dependency to the block interface. In particular, it was also designed to fit well with the Object-Based Devices (OBD) model. A Proboscis implementation at the OBD level could serve as a storage subsystem for software such as the ActiveScale File System [6], that relies on network-attached secure disks [12].

the Proboscis framework, so the network extension is free to optimize communication for the specific network technology. A client gains access to a remote disk through a proboscis through the extension `os2prob` that acts as an interface between the block device abstraction of the operating system and the Proboscis framework. In our example, `os2prob` is connected to the named proboscis `mirrored_disk`.

When the OS performs operations on the block device handled by the proboscis, the operations are converted into *commands* by `os2prob`. `os2prob` issues these commands through a socket and will receive a completion notification asynchronously, i.e., the communication semantic used is asynchronous procedure calls. A command travels through the proboscis to its destination(s) and returns by the reverse path(s). In case of a write in our example, `os2prob` will issue a write command to the `mirror` extension, which will clone it and issue concurrent commands to the two remote disks. When both have returned completion notifications to `mirror`, it will return a completion notification to `os2prob` which in turn will finish the operation posted by the OS.

The extensions can be viewed as interpreters of the command stream along the data access paths. To place as few restrictions as possible on the processing of the command flow, an extension must implement a non-blocking, reentrant event-driven state machine. The primary source of events are arrivals of Proboscis commands and completion notifications, but the extensions can define additional events themselves. Within a given proboscis, it can be necessary for an extension to maintain local data; we therefore distinguish between the extension itself (defined by a code module) and an *instance* of an extension.

Except special cases—end extensions that interact with the operating system layers (`os2prob`, `prob2os`) and networking extensions—we have three categories of extensions: *in-line*, *scatter* and *gather* extensions. An *in-line* extension has exactly one root socket and one tip socket; it can be interposed for monitoring or modifying the parameters of a command. A *scatter* extension (e.g. `mirror`) has one root socket and n ($>$1) tip sockets; it is typically employed to implement a striping/redundancy scheme. Finally, *gather* extensions (like `share`) have n ($>$1) root sockets and one tip socket; they merge several client paths into a single path in the device direction. The Proboscis framework requires that all extensions must have one side with exactly one socket to allow reconfiguration operations to take place anywhere in a proboscis.

3. Reconfiguration

In this section, we first motivate our view of reconfiguration as an important form of storage management. Then we describe our generic approach to reconfiguration in the Proboscis framework, followed by a couple of concrete examples.

3.1 Storage Management by Reconfiguration

A large set of management tasks in a storage system can be expressed as modifications to either the structure or the processing of the data access paths. We identify the following categories of reconfiguration actions useful for managing a cooperative storage system for clusters:

Code updates can be useful for a number of reasons : to fix security holes or bugs, to provide a better implementation of a given module, or to add new features to it (e.g. more control over its parameters).

Modifications of non functional attributes allow to adapt the system's operation to external constraints. For example, clustered multimedia servers need to dynamically adjust the QoS requirements of their different I/O streams. Another case for such a functionality concerns the networking protocols and hardware used for storage I/O. As current interconnection technologies support hot-plugging [29], administrators can add network interface cards (NICs) without stopping the cluster nodes and the services they run. The "reverse scenario" is equally plausible. Most clusters' nodes are equipped with several NICs: a cheap one (usually Fast Ethernet) for administration purposes and one or several others for high-performance communications. During a maintenance intervention on the high-speed equipment, the low-end networking infrastructure could provide a continuity of service for the storage traffic. However, this feature is only useful if the storage system can dynamically switch from a given networking protocol/device to another.

Modifications of the system's architecture and/or operation concern the core role of the cluster storage infrastructure, i.e. enforcing a given data distribution pattern (for fault tolerance and load balancing) and routing I/O requests to and from the appropriate devices. Useful measures include the addition of shared disks and the redistribution of the data (to enhance the fault tolerance level, cope with changes in I/O load, or rebuild a disk after a crash).

Dynamic instrumentation can help administrators detect anomalies and tune the performance of the system. The dynamic approach is more advantageous than a static one since it avoids monitoring overheads that are useless during normal operation. Moreover, new probes can be developed and used long after the system was initially started.

It is noteworthy that some of these reconfiguration actions may require or benefit from another kind of reconfiguration action. For instance, the modification of the system's architecture may involve the introduction of new features and imply a code update. Similarly, dynamic profiling may encourage the upgrade of some code modules.

3.2 Reconfiguration in the Proboscis framework

In the Proboscis framework, the reconfiguration of a storage system is performed by modifying the distributed data structure. A modification can either extend, reduce or replace parts of this data structure. This section presents an overview of the mechanisms provided by the Proboscis Framework to deal with the reconfiguration needs identified in 3.1. These mechanisms heavily benefit from the component-based nature of Proboscis. They also rely on techniques described by Soules et al in [31], where four capabilities are identified to implement interposition and hot-swapping of components:

1. *The system must be able to identify and encapsulate the code and data for a swappable component.*
2. *The system must be able to quiesce a swappable component such that no references are actively using its code and data.*
3. *The system must be able to transfer the internal state of a swappable component to a replacement version of that component.*
4. *The system must be able to modify all external references (both data and code pointers) to a swappable component.*

Let us start by describing how the Proboscis architecture matches these requirements. The component model used by Proboscis intrinsically satisfies criterion 1[3]. Criterion 2 can be matched quite easily[4]; since Proboscis uses only one processing thread per node and extensions are event-based, it is straightforward to determine when a component is not 'actively used' and to block the next call to it. When suspending execution of an instance, arriving commands are simply put on hold on a per instance queue. The state transfer facility required by criterion 3 can be achieved without much burden on the extension developers because extensions are assigned simple roles and, as a consequence, do not store a high quantity of state information. Problems regarding criterion 4 are solved using indirection pointers both for the bindings between extensions and for the methods exported by an extension.

3.2.1 Generic reconfiguration mechanisms in the Proboscis Framework

We will now revisit the reconfiguration actions from Section 3.1 and describe how they are supported in the Proboscis framework.

[3]As the current prototype is based on Linux loadable kernel modules written in C, we rely on the developers respect to the programming model. Metacompilation techniques could help enforcing these rules.

[4]In particular, without resorting to the 3-phase protocol in [31].

Code updates are implemented at the granularity of a Proboscis extension and rely on the hot-swapping mechanism described above. In any system supporting dynamic loading of kernel modules, it is possible to replace a given instance with a new version. On every node, the framework manages the library of installed extensions and keeps track of their respective version numbers.

Most of the **modifications of non functional attributes** are handled with the definition of new Proboscis commands. For instance, it is possible to issue a command (through a proboscis) to a disk sharing extension (`share` on figure 1) in order to change the scheduling policy regarding its different client streams or its threshold for batching disk requests. Only the code of the concerned extension needs to take this type of command into account. The other extensions apply a generic routing mechanism and do not need to know more about these specific commands. Therefore, if the code of an extension is updated to provide a finer control over its runtime attributes, the code upgrade will not impact the other kinds of extensions. The capability of dynamic switching between two networking technologies, which is not managed as the other operations on non-functional attributes, is detailed in section 3.3.1.

Modifications of the system's architecture and/or operation can be achieved by any of the following three kinds of reconfiguration actions: (1) adding or removing a proboscis to a gather extension, (2) replacing a proboscis with another proboscis, and (3) modifying a proboscis by extending or compressing its composition. Actions 1 and 2 are implemented using standard `create` and `destroy` commands provided by the framework. The third kind of action requires the use of two special reconfiguration commands: `insert` and `remove`. An `insert` (respectively `remove`) command allows the injection (resp. eviction) of one or several instances into (resp. from) an already deployed proboscis path. An insertion can replace a joint by a series of instances or take place at the end of a proboscis (e.g. a path to a new disk is grafted to a gather extension such as `mirror`). The same distinction applies for a removal.

In most cases, `insert` and `remove` operations require some form of coordination with the regular I/O traffic, which is achieved as follows. When such a reconfiguration command is issued, it raises a synchronization barrier covering the extensions affected by the reconfiguration command: it waits for the termination of in-transit commands (mostly `read` and `write` commands) and queues new commands. For a reconfiguration action that impacts a portion of a proboscis shared by several root nodes, the barrier is raised at the level of the sharing instance. Otherwise, the barrier is raised at the level of the first instance hit by

the command (this is typically an `os2prob` instance).

Once all the awaited commands have been acknowledged, the reconfiguration command is issued to the extension and routed through the proboscis path. When the `insert` (resp. `remove`) command reaches the instance where the operation should start, a child `create` (resp. `destroy`) command is used locally to modify the structure of the path. On its issuing way, the `create` (resp. `destroy`) commands only performs checks[5]; the bindings between instances are only updated on the way back of the command, once all the verifications have succeeded. In this way, the reconfiguration is only performed if it is guaranteed to succeed and a proboscis path can not end with an inconsistent composition.

When the reconfiguration command reaches the initial synchronization barrier, the latter is deactivated and the queued I/O requests are processed. This reconfiguration scheme actually implies a disruption, in the sense that regular I/O commands are blocked. However, the traffic interruption is transparent (except the greater latency perceived for the blocked I/O commands) for the other instances of the targeted proboscis and *a fortiori*, for the upper layers of the system (block device driver, file system, applications). Moreover, the disruption is very short and does not impose an exorbitant performance penalty (as exposed in section 4).

This method was adopted because it greatly simplifies the implementation of the reconfiguration process and does not impose an extra burden of the extension developers. Indeed, without a synchronization barrier, commands could have a return path crossing the modification point. This is not a trivial problem since any Proboscis command carries information (such as a stack and annotations) that can be modified by the instances it flows through. Enforcing the coherence of a command's contents in this context would result in a complex implementation for a modest potential performance gain.

Finally, **dynamic instrumentation** is based on the techniques developed for the two previous points. Pre-defined monitoring/profiling facilities in the code of an extension can be switched on or off with extension specific commands. Probes can also be injected in the system when necessary using `insert` and `remove` commands.

3.2.2 State server

The state server (SS) is a central entity that maintains a coherent view of the global state of the storage system. In particular, it maintains metadata (*"skeletons"*) about all the instances deployed throughout the cluster, including their bindings to other extensions and the version of the associated binary. The SS also provides a name service (mentioned in section 2) and assigns global identifiers and version numbers to the proboscis paths to detect potential outdated metadata on the participating nodes. Cluster nodes running the Proboscis Framework must contact the SS prior to executing any operation that is non I/O related: path creation, destruction, (re)naming or reconfiguration. In addition to keeping the metadata up to date, this allows the synchronization of concurrent reconfiguration operations (through the grant of locks associated to proboscis paths) and to provide atomicity for reconfiguration actions that imply the issuing of several consecutive commands (like in section 3.3.1, for instance).

The SS is also in charge of determining (with a heartbeat protocol) that a node has crashed (or is unreachable) and should not be used by the storage infrastructure anymore (and that locks granted to it must be taken back). Finally, the SS maintains a list of spare disks and nodes that can be used by the cluster storage system to cope with a disk failure or any other need (the list can be dynamically updated by an administrator).

The state server is intended to be run on a server machine accessible to all the cluster nodes. Having a centralized service like the state server raises concerns about fault tolerance and performance bottlenecks. However, both the number of proboscises in a cluster and their configurations change at a low rate, so fault tolerance can be supported in a number of ways, e.g., with a log file on a shared remote disk and a master/slave protocol with a backup node. Performance bottlenecks are also unlikely due to the low rate of operations. In case of a server crash, the recovery will thus take a limited time and only affect the latency of the reconfiguration requests (I/O requests do not involve the state server).

3.2.3 Towards an autonomic storage system

Proboscis commands can be generated in several ways. The most obvious and common way is to issue commands from the top of a proboscis path. For I/O requests, this is handled by the `os2prob` instance that interacts with the I/O layer of the operating system. For other kinds of requests (creation/destruction/reconfiguration of a path), a user-space utility interacts with the Proboscis infrastructure kernel module (via `ioctl` calls).

However, commands can also be generated by any instance within a proboscis. This feature is necessary for I/O commands (for instance, a RAID extension splits a parent write command into several children commands to the different disks involved in the transfer) but can also be exploited for reconfiguration actions. In particular, a self-reconfigurable logic can be integrated into the code of an

[5] More precisely, a `create` command checks that the allocations and initializations of new instances are successful while a `destroy` command checks that targeted instances are in a state compatible with removal.

extension. While providing its core functionality, an extension can notice the occurrence of a typical condition for reconfiguration (e.g. a disk failure or a low performance threshold being reached). When such a condition is verified, the instance can initiate a reconfiguration process. It starts by contacting the state server to request the necessary lock(s) on the concerned path. The request includes the version number of the proboscis' skeleton stored by the instance; if it is outdated, the request is rejected by the SS and the updated skeleton is provided to the instance. Once its local metadata has been updated (which is necessary since the routing for a reconfiguration command is determined from the source), the instance can try to grab the lock again.

The request can also be rejected because the concerned lock(s) has (have) already been granted. In this case, this means that another entity (a human administrator or another instance) has already diagnosed a case for reconfiguration and has initiated the appropriate action. The extension's logic can then either decide to abort its initiative or schedule a new check for later (followed, if necessary, by a new request to the SS). When a lock request succeeds, the instance can start a reconfiguration action, implying the generation of one or several Proboscis commands. Once the action is done (or has failed), the instance metadata is used to update the information held by the SS and the locks are released.

Proboscis commands can also be issued through a proboscis in the name of the state server, which can communicate with a local daemon deployed on all the cluster nodes. As the state server has a global view of the system, it can be used to search for all the instances/proboscises satisfying a given criterion and automatically generate and issue the necessary reconfiguration commands to the concerned targets. Thus, an administrator can use a console on the state server to address high level requests such as "`switch network connection in proboscis {1,2,3} from NW=(SCI) to NW=(GigabitEthernet, TCP)`" or "`update extension code for EXT=mirror on proboscis {*} to VERSION=(1.6)`".

Furthermore, we think that this model allows to progressively enrich the system (in the extensions and at the state server level) with an autonomic logic. In the future, the SS could automatically inject probes into the system to extract performance results (of both the storage system and the applications running on disk nodes) and decide on the appropriate measures to take, detect failures, migrate and/or rebuild data and signal the human administrators that a disk must be replaced. This is definitely a long term objective but we think that this is an achievable goal. Indeed, the context of storage systems has relatively clear high-level requirements (at least for criteria such as space and level of redundancy, performance specifications and expectations are harder to express [10]) and that Proboscis abstractions (three main kinds of components: in-line, scatter, gather and a limited set of extensions, with a coarse-grained functionality) are simple enough to be automatically managed.

3.3 Reconfiguration Examples

This section reports on two use cases of the reconfiguration features: the dynamic network switching capability mentioned in section 3.1 and an auto-adaptable, self-recovering storage system based on mirrored disks. The first example illustrates how the mechanisms described in 3.2.1 help providing a continuity of service to applications; the second example sheds light on the semi-autonomic features built on our framework. Due to the lack of space, the interactions with the state server are not systematically listed.

3.3.1 Dynamic network switching

To illustrate this functionality, we consider the migration of a proboscis from one networking protocol/technology (a specialized protocol over SCI [20]) to another (TCP/IP over Ethernet). Initially, the proboscis is composed as in step 1 on figure 2. For simplification purposes, the whole path is not represented on the figure. Moreover, the role of instances `A` and `B` does not matter in this context (for example, `A` and `B` could be respectively `os2prob` and `prob2os` instances or `mirror` and `share` instances).

First, a "Network Switch" (`NWS`) command is issued to instance `A`. The command parameters include the IDs of the `SCI` instances to be removed and the necessary information to deploy the new networking extensions. In reaction to the `NWS` command, instance A sends two `insert` commands[6]: a `nws` (gather) instance is inserted (in tip direction) "behind" `A` and a `share` instance is inserted (in tip direction) "in front of" `B`. Then, the `nws` instance issues a third `insert` command to deploy the new (TCP/IP) networked path, which leads to state 2 depicted on figure 2. When this new connection is established, the `nws` and `share` instances switch their default route to the TCP path, i.e. all the new commands will be routed through the `TCP` instances while the `SCI` ones will handle the acknowledged commands that were issued through them.

At some point, the `nws` instance will determine that all the SCI related commands have been acknowledged and will trigger the removal of the SCI instances, followed by those of `nws` and `share`. Finally, `A` will acknowledge the `NWS` command and the proboscis will end up with having the `SCI` instance replaced by the `TCP` instance as shown in the third and final step of figure 2.

[6] The instance's reaction to the `NWS` command does not have to be integrated into the extension's code: it is "inherited" from the generic mechanisms provided by the framework.

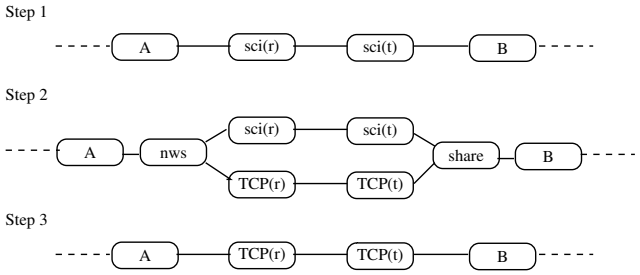

Figure 2. Main steps of a dynamic network switching within a proboscis

3.3.2 Auto-adaptable disk mirroring

Most cluster storage infrastructures rely on redundancy schemes such as RAID 5 [25], Chained Declustering [16] or Orthogonal Striping and Mirroring (OSM) [17]. Though these methods have various performance merits according to the characteristics of the workload they are exposed to[7], they share a common weakness in that they can only tolerate a single disk failure. This is likely to be unacceptable for large clusters with common off-the-shelf components. To address this issue, the DRAID project [8] defined a new distributed data layout that provides a tunable redundancy level. However, the performance is limited and the rebuild phase is complex. As the price/capacity ratio for disk drives (and the price/bandwidth ratio for networks) continue to drop while administration costs rise with increasingly large and complex systems, we think that approaches based on mirroring like RAID 10 [25] become more and more appealing, given their good performance and their ease of implementation and management (notably for disk rebuild). Thus, we consider the following example based on mirrored disks a realistic case-study.

This example is based on the layout described in section 2 and in figure 1. To keep the description simple, we only show a 2-2 (client nodes - disk nodes) configuration but the exposed principles can be extended to any n-m setup in a straightforward manner.

Let us assume that the disk exported by "disk node 2" has crashed. This event is detected by the `mirror` instances on the clients when a command returns an I/O error condition. As a consequence, the `mirror` instances initiate the removal of their path to the failed disk. Because of these `remove` operations, the mirrors must contact the state server. In this way, they will inform the SS that the concerned disk should be replaced/examined by human intervention. Moreover, while competing for the locks associated with `shared_disk_2`, one of the `mirror` instances (`mirror_1` on "client 1" in our case) is elected as the master of the reconfiguration process.

The master first retrieves by the state server a named proboscis describing the access to a spare disk. Second, the master inserts a `copy`[8] instance in front of the remaining active `prob2os` instance (on "disk node 1") and then connects the `copy` instance to a shared proboscis (`share_disk_3`) deployed on the new node ("disk node 3") on that occasion. We refer to the connection between the copy instance and `share_disk_3` as the "copy path".

At this point, the configuration of the storage system corresponds to the one depicted in figure 3. A `start_copy` command is sent to the `copy` instance to trigger the cloning of disk 1. During this process, client writes are synchronized between disk 1 and disk 3. Having the `copy` instance just in front of the `prob2os` instance allows for monitoring all the traffic (from all clients) to the disk and helps to deal with the synchronization of the new mirror.

Once the whole disk (partition) has been copied on disk 3[9], the master `mirror` instance connects to `share_disk_3` and sends a special `write_sync` command[10] to the two disk paths. When this `write_sync` is received on `share_3` from the two incoming connections (the new path and the copy path), the master is considered synchronized with the new disk. When the acknowledged `write_sync` command reaches the copy instance, the latter stops forwarding write requests from "client 1" to "disk node 3" (but keeps forwarding writes from "client 2", which is not synchronized with the new mirror yet).

Upon reception of the acknowledged `write_sync`, the `share_2` instance notices that the master is done with the reconfiguration and sends an `update_mirror` command to all the other clients[11] by means of a *reverse broadcast*. In its turn, "clients 2" connects to the new disk using the same synchronization protocol. When `share_2` detects that all the clients are synchronized, it initiates the removal of the copy path and the `copy` instance.

This reconfiguration mechanism impacts only the code of three extensions: `mirror`, `share` and `copy` and can be used to clone multiple mirrors at the same time[12]. With modifications to the `prob2disk` extension, it could also be adapted to other data distribution schemes, including hybrid schemes that dynamically mix RAID 1 and RAID 5 (to optimize the performance/space ratio) [26]. In addition

[7] Distributed RAID 5 also raises synchronization issues that complicate its implementation and limit its performance [1].

[8] If there are several remaining active mirrors, the choice is up to the master. As the skeleton metadata held by the master is up to date (since its interaction with the SS), the latter has all the necessary information for the routing of the `insert` commands.

[9] i.e. when the master receives the acknowledged `start_copy` command

[10] This command is special in the sense that it carries not data to be written; it is only used as a synchronization signal.

[11] There is only "client 2" in our case.

[12] This is useful to deal with the crash of a node that exported several disks/partitions.

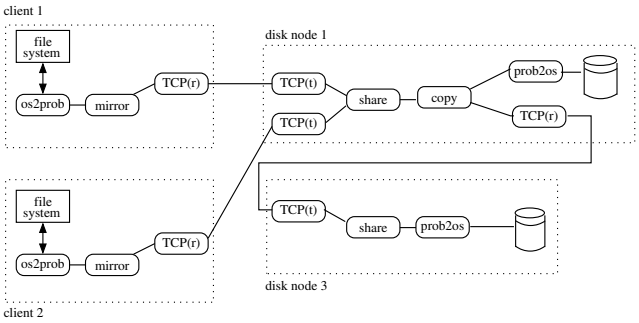

Figure 3. Example showing a mirror disk being rebuilt

to self-recovery, this mechanism could be used for automated performance tuning, e.g. if a disk or a network connection provides disappointing performance, a "preventive" data migration can be performed.

4. Implementation

In this section, we describe our prototype implementation of the two examples presented in 3.3 and we estimate the cost of performing reconfiguration actions, i.e. the time necessary to handle reconfiguration commands and their impact on the performance of the I/O subsystem.

The Proboscis framework is implemented as a set of loadable kernel modules for Linux (on x86 and Itanium platforms). The results presented in this section were gathered on the following test configuration: the nodes were equipped with an Athlon 1800MP processor, 1 GB of DDRAM, an AMD 760MP chipset, a Maxtor 6l08l0J4 IDE (slave) disk, a Broadcom 5701 Gigabit Ethernet adapter and a Dolphin D330 SCI adapter. Both networks were switched with respectively an HP Procurve 2724 and a Dolphin D535 model. The machines ran Linux kernel version 2.4.20.

4.1 Example 1: dynamic network switching

We measured the average time needed to perform the reconfiguration steps described in 3.3.1 with and without concurrent I/Os. The I/O stream used in the first case consisted in an intensive burst of 128 kB requests resulting in a 1 GB write transfer. We chose such an intensive workload because it represents a "worst case scenario" in terms of sensibility to the intrusiveness of concurrent operations.

For remote operations (e.g. the insertion of `share` or the removal of the old network connection), we measured an average time of 340 μs while we obtained 35 μs for local operations (e.g. the insertion of `nws`). Seen from the administration console, the whole operation completes in an average of 2.4 seconds (for a single proboscis - but concurrent reconfiguration of distinct proboscises can be performed in parallel). These results did not show any significant variation due to the concurrent I/O traffic.

We also measured the impact of the reconfiguration actions on the latency and the throughput of the data transfers. Determining the influence of the dynamic network switching on the performance is not obvious since the characteristics of the two networking facilities are not identical: we switch from an SCI-optimized communication layer to TCP/IP over Gigabit Ethernet. To overcome this bias, we adopted the following methodology[13]. First, we run the benchmark (i.e. the I/O stream) several times without any concurrent reconfiguration, to determine the average performance for a given networking setup. Then, we start the benchmark (on the SCI network) and wait for a fixed time before starting the reconfiguration protocol. In this way, we know exactly how much time was spent with each kind of network and we can determine the slowdown caused by the reconfiguration protocol[14].

An average 260 requests (3% of the total number of requests) are delayed because of the synchronization barriers raised by the reconfiguration commands. Delayed requests show a 30% increase in average latency compared to the regular case but the maximum latency observed for delayed requests is lower than the peak latency. Regarding throughput, we determined an average 11% slowdown caused by the reconfiguration.

4.2 Example 2: self-recoverable mirroring

We started by ensuring that, without concurrent I/Os, the copy operation can be performed at the full (target) disk speed. Then, we ran a simple benchmark to estimate how the client requests and the copy process impact on each other. The setup corresponds to the one described in figure 3, i.e. one or two clients accessing the remaining mirror while a new one is being created. The clients issue an intensive burst of sequential I/O requests while the mirror is being cloned. For a write (respectively read) test, each client keeps writing (resp. reading) one half of the accessed 10 GB partition.

These simple tests lack realism but we consider them close to a "worst case scenario" since synchronized writes are perpetually issued and no load balancing can be performed for reads (until the new mirror is up). Moreover, the intensive nature of the I/O streams compensate for the low number of clients.

We obtained throughput slowdowns in the ranges of

[13] N.B.: We start all experiments with cold caches.

[14] As exposed previously, the time taken by the reconfiguration phase (a few milliseconds) is negligible compared to the duration of the benchmark (several tens of seconds).

[11%-45%] and [6%-52%] for respectively the client streams and the rebuild process. The measured latencies for reconfiguration commands were similar to those of section 4.1.

The current implementation of the `copy` extension is very basic: it copies the source disk in a sequential manner and synchronizes write commands. The time measured for the cloning operation is thus an upper bound. We expect to reduce this time with a finer (and dynamic) tuning of the `copy` extension parameters, the use of hints that batch synchronized writes with the I/O traffic, or file-system aware compression schemes [15]. Once these optimizations are implemented, we intend to perform more realistic benchmarks (in terms of workload and client number).

4.3 Summary

The above experiments do not provide a complete performance evaluation but show that our approach is viable since even rough implementations of the reconfiguration extensions induce an relatively acceptable overhead under a high I/O load.

5. Related Work

Our work is partially motivated by the observation that existing cooperative storage systems for clusters (CSSC) [3, 17, 8] lack configurability and, *a fortiori*, on-line reconfiguration features.

Most CSSC only offer a limited control of their properties. This choice is often restricted to the static configuration of the redundancy scheme and a list specifying a few hot spare devices and some slave nodes that can replace a failed master.

An noticeable exception is Petal [21], whose core concept of *virtual volumes* (close to the one of virtual memory) greatly simplifies dynamic actions such as the management of the storage pool (addition or removal of devices), replacement of the data redundancy scheme and efficient snapshots. Proboscis focuses on lower level concerns (flexible access to storage devices) and could be used as the foundation for implementing a Distributed Virtual Volume Manager like Petal.

Dynamic management of the storage pool can also be dealt with at the logical volume manager level (LVM), when this does not interfere with the architectural constraints of the CSSC (frameworks like Petal deal themselves with volumes and can hardly cooperate with conventional LVMs). Some LVMs (such as Veritas' VxVM [33] and HP-UX' LVM [24]) support on-line migration. The Aqueduct tool [23], enforces a human-specified quality-of-service (QoS) for I/O streams related to foreground applications while performing a data migration. A recent framework for building unobtrusive disk maintenance applications (such as a snapshot-based backup, a buffer cache cleaner or a layout reorganizer) has also been proposed [32].

Such tools can be useful for cluster management but they do not solve by themselves the problems of the CSSC layer, which requires its own hooks for simplified and non-disruptive administration. Besides, the logic developed in [23, 32] could be interfaced with Proboscis (e.g. to dynamically adjust the attributes of an instance) to provide a fine control over the maintenance operation.

Recent years have seen the emergence of a new vision for the design of computing systems: *Autonomic Computing*, i.e. systems that can manage themselves given high-level objectives from administrators [19]. The *Self-* Storage* project (S*S) [10] at Carnegie Mellon University can be described as an application of the goals and concepts of Autonomic Computing to the context of storage systems. The S*S architecture is based on the use of networked "intelligent" storage bricks [9, 18] and distinguishes several entities: a hierarchy of supervisors controls and adjusts the operation of workers (the cooperative storage bricks) while (external) clients access data through command routers to the bricks. A prototype is currently being deployed with an object-based store exported to clients with NFS servers.

We share much of the ideas of the S*S vision. However, our approaches do present some differences. First, we focus on intra-cluster data-sharing and thus do not study routers to external clients. More importantly, if we agree that radical architectural and management changes will be mandatory to deal with increasingly complex systems in the long run, we think that, transitional solutions are still necessary since progress (development of the autonomic logic embedded into the components, massive adoption of new interfaces like Object-Based Devices...) is likely to take time. In the meantime, semi-automated tools that ease frequent tasks (like console broadcast or disk cloning software) can still be of a great help. Our framework allows to build such tools while performing a progressive integration of embedded intelligence in the system.

6. Conclusion

In this paper, we have described how a cooperative cluster storage system can perform storage management by inserting and removing code modules at runtime within a framework of explicit storage access paths. This allows management tasks to be carried out on the nodes in the cluster where it is most efficient from a storage traffic perspective and allows reconfiguration of the storage system without taking it offline.

Experiences from our prototype implementation show that it is possible to perform management tasks such as network migration and rebuilding failed disks in a mirror

with little disruption for the storage users. Furthermore, the framework aids the implementation of such tasks and allows them to be carried out with little human intervention.

Future efforts will work towards integrating the management of storage paths into disk maintenance applications and to building code modules with self-tuning capabilities.

References

[1] K. Amiri et al. Highly Concurrent Shared Storage. In *Proceedings of the 20th International Conference on Distributed Computing Systems (ICDCS)*, April 2000.

[2] T. E. Anderson et al. Serverless network file systems. In *Proceedings of the 15th ACM symposium on Operating Systems Principles*, 1995.

[3] C. R. Attanasio et al. Design and Implementation of a Recoverable Virtual Shared Disk. Technical Report Research Report RC 19843, IBM T. J. Watson Research Center, Yorktown Heights, NY, USA, 1994.

[4] P. T. Breuer et al. The Network Block Device. *Linux Journal*, (73), May 2000.

[5] P. Carns et al. PVFS: A parallel file system for linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, Atlanta, GA, USA, 2000. USENIX Association.

[6] P. Corp. The ActiveScale File System, 2004. See `http://www.panasas.com/activescaleos.html` for details (2004-06-23).

[7] O. Cozette et al. READ2: Put disks at network level. In *Proceedings of the 3rd International Symposium on Cluster Computing and the Grid*, Tokyo, Japan, May 2003.

[8] A. Di Marco et al. Using a Gigabit Ethernet Cluster as a Distributed Disk Array with Multiple Fault Tolerance . In *Proceedings of the 28th Annual IEEE International Conference on Local Computer Networks*, October 2003.

[9] S. Frolund et al. FAB: Enterprise Storage Systems on a Shoestring. In *Proceedings of the 9th USENIX Workshop on Hot Topics in Operating Systems*, May 2003.

[10] G. R. Ganger et al. Self-* Storage: Brick-based Storage with Automated Administration. Technical Report CMU-CS-03-178, Parallel Data Lab - Carnegie-Mellon University, August 2003.

[11] S. Ghemawat et al. The Google file system. In *Proceedings of the 19th ACM symposium on Operating Systems Principles*, October 2003.

[12] G. Gibson et al. A Cost-Effective, High-Bandwidth Storage Architecture. In *Proceedings of the 8th Symposium on Architectural Support for Programming Languages and Operating Systems*, 1998.

[13] J. S. Hansen et al. Using Idle Disks in a Cluster as a High-Performance Storage System. In *Proceedings of IEEE Cluster 2002*, pages 245–254, September 2002.

[14] J. H. Hartman et al. The zebra striped network file system. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 29–43, 1993.

[15] M. Hibler et al. Fast, Scalable Disk Imaging with Frisbee. In *Proceedings of the 2003 USENIX Annual Technical Conf.*, San Antonio, TX, USA, 2003.

[16] H. Hsiao et al. Chained Declustering: A New Availability Strategy for Multiprocessor Database Machines. In *Proceedings of the Sixth International Conference on Data Engineering*, 1990.

[17] K. Hwang et al. Orthogonal Striping and Mirroring in Distributed RAID for I/O-Centric Cluster Computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(1), 2002.

[18] IBM Almaden Research Center. Collective Intelligent Bricks, August 2003. See `http://www.almaden.ibm.com/StorageSystems/autonomic\_storage/CIB/` (2004-04-22).

[19] IBM Corporation. Autonomic Computing: IBM's perspective on the State of Information Technology, October 2001. See `http://www.research.ibm.com/autonomic/manifesto/` (2004-04-12).

[20] IEEE. *IEEE Standard for Scalable Coherent Interface (SCI)*. 1992. Standard 1596-1992.

[21] E. K. Lee et al. Petal: Distributed Virtual Disks. In *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, USA, October 1996.

[22] P. Lombard et al. nfsp : A Distributed NFS Server for Clusters of Workstations. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS 2002)*, Fort Lauderdale, FL, USA, April 2002.

[23] C. Lu et al. Aqueduct: online data migration with performance guarantees. In *Proceedings of FAST '02*, January 2002.

[24] T. Madell. *Disk and File Management Tasks in HP-UX*. Prentice Hall, 1997.

[25] P. Massiglia. RAID for Entreprise Computing. A Technology White Paper. Technical report, Veritas Software Corporation, 2000.

[26] M. Pillai et al. A High Performance Redundancy Scheme for Cluster File Systems. In *Proceedings of IEEE Cluster 2003*, Hong Kong, December 2003.

[27] K. Preslan et al. A 64-bit, Shared Disk File System for Linux. In *Proceedings of the Seventh NASA Goddard Conference on Mass Storage Systems and Technologies*, San Diego, CA, USA, March 1999.

[28] A. Pruscino et al. Oracle RAC 10g: The Fourth Generation. whitepaper, September 2003. Oracle Corp.

[29] R. Recio. Server I/O Networks Past, Present, and Future. In *Proceedings of the ACM SIGCOMM Workshop on Network-I/O Convergence*, 2003.

[30] F. B. Schmuck et al. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proceedings of FAST '02*, 2002.

[31] C. Soules et al. System Support for Online Reconfiguration. In *Proceedings of the Usenix Technical Conference*, San Antonio, TX, USA, June 2003.

[32] E. Thereska et al. A Framework for Building Unobtrusive Disk Maintenance Applications. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST '04)*, San Francisco, CA, USA, March 2004.

[33] Veritas Software Corporation. Veritas Volume Manager. See `http://www.veritas.com/van/products/volumemanagerwin.html` (2004-04-13).