Data ONTAP GX: A Scalable Storage Cluster

Michael Eisler, Peter Corbett, Michael Kazar, and Daniel S. Nydick
Network Appliance, Inc.

J. Christopher Wagner
IronPort Systems, Inc.

Abstract

Data ONTAP GX is a clustered Network Attached File server composed of a number of cooperating filers. Each filer manages its own local file system, which consists of a number of disconnected flexible volumes. A separate namespace infrastructure runs within the cluster, which connects the volumes into one or more namespaces by means of internal junctions. The cluster collectively exposes a potentially large number of separate virtual servers, each with its own independent namespace, security and administrative domain. The cluster implements a protocol routing and translation layer which translates requests in all incoming file protocols into a single unified internal file access protocol called SpinNP. The translated requests are then forwarded to the correct filer within the cluster for servicing by the local file system instance. This provides data location transparency, which is used to support transparent data migration, load balancing, mirroring for load sharing and data protection, and fault tolerance. The cluster itself greatly simplifies the administration of a large number of filers by consolidating them into a single system image. Results from benchmarks (over one million file operations per second on a 24 node cluster) and customer experience demonstrate linear scaling.

1 Introduction

File storage is divided between local file systems and network file systems. As networks have become faster and more reliable, network file systems have become an important aspect of most organizations IT infrastructure. Typical applications include home directories, databases, email, and scientific and technical computing.

The widespread deployment of network file systems has led to the development of specialized file server solutions, commonly referred to as Network Attached Storage (NAS). NAS systems, generically called **filers**, have typically been monolithic systems, with a single or dual controller or head, fronting a large amount of disk. Such systems often support virtualization, allowing the aggregated disk storage to be divided into a number of virtual volumes, allowing the virtual volumes to be presented through a number of virtual servers, all hosted by the same filer hardware.

The limits of this approach are obvious. As the number of client machines attached to networks increases, the number of filers must increase commensurately, or the filers will become overloaded. A filer is based on similar hardware to a client, and so, the only way for a filer to ”keep up is to either add more filers, or to increase the performance of the filer. The second option is expensive; specialized hardware solutions seldom maintain a performance advantage over commodity hardware.

But adding more filers has the disadvantages of being complex to administer, disallowing opportunity for load balancing and sharing among the filers, and requiring the clients to mount a large number of different filer volumes.

We decided that the best solution to this problem is to cluster a number of individual filers to form a single file server. For this purpose, we developed GX, which leverages the existing ONTAP-7G architecture [NetApp], but adds a switched virtualization layer just below the client-facing interfaces. This allows the storage of a large number of filers to be presented as a single shared storage pool. The key features provided are scalability, through the ability to add filers to the cluster, location transparency of data within the cluster, an extended namespace that can span multiple filers, increased resiliency in the face of failures, and simplified load and capacity balancing.

2 Related Work

GX draws on much previous work. It uses a remote file system switching mechanism inspired by the Virtual File System (VFS) [Kleiman]. GX supports both NFS

[Sun] and CIFS [SNIA]. GX provides many benefits which are drawn from the Andrew File System [Howard], and its commercial successors AFS [Campbell] and DFS [Kazar1990]. The AFS namespace connected multiple disparate AFS servers, each of which stored a **cell**. Each cell could store multiple volumes, and volumes were linked to each other through internal mount points, with the linkage and location of the volumes defined by the Volume Location Database. AFS also provided benefits such as location transparency, and the ability to load balance beneath a single client mount point.

However, AFS and DFS required special client code. The experience of AFS and DFS in trying to establish a client footprint was one of the main observations that motivated the use of NFS and CIFS as the client access protocols in GX. The goal of GX was to provide the benefits of AFS and DFS, while providing client access through the widely deployed NFS and CIFS protocols.

The GX architecture is inspired by that of Spinnaker Networks [Kazar2002]. Spinnaker was acquired by Network Appliance in 2004.

Frangipani is a SAN-type file system based on a distributed lock manager coordinating accesses from a collection of file system clients to a shared virtual disk [Thekkath]. We decided against building a SAN file system because of concerns about the overhead of distributed lock management in workloads with read/write data sharing, or high volumes of meta-data updates of any kind. We were also concerned about the size of the failure domain in such an architecture, where a bad piece of hardware could cause almost unbounded damage.

Slice has a goal to distribute the directory operations across many servers without partitioning the namespace into volumes, which avoids user visible mount points, re-partitioning volumes if volume loads grow at uneven rates, and avoids the issue of hard link and rename crossing volume boundaries [Anderson]. Slice distributes every object separately, based on a hash of its file name. Thus name lookups distribute well across the cluster. Hard link creation, rename, and file removal require a two phase commit across servers. We ruled out such an approach due to the overhead of distributed transactions.

3 Architecture

The architecture of GX addresses a number of key challenges in clustered NAS service.

First, we wanted GX to provide horizontal scaling, so that a cluster could grow over time to provide additional storage and processing capabilities for the namespace exported by GX.

Second, we wanted location independence, so that data could be reconfigured dynamically while the system was operating.

Third, we wanted to abstract the externally visible notion of a ”server away from the physical hardware. This would allow us to provide multiple virtual servers within the clustered system, so that the actual set of physical resources dedicated to a particular name space can be chosen to match what is required for this name space, and does not have be reserved in any special fixed size units, such as entire disks or network interfaces.

Fourth, we wanted to pay as little as possible for these features.

3.1 Overview

The GX architecture is a high level switched architecture, where network file system requests are received by a file server s front end, mapped into one or more simple file system requests, and then transferred over a cluster fabric to the server that stores the data.

Data is stored in **volumes**, which are file system sub-trees consisting of an inode file with a root inode and a set of directories and files contained under that root inode. An **aggregate** is a collection of volumes, which can be thought of as a virtualized UNIX disk partition.

A **namespace** is composed of volumes, joined together by **junctions** which are entries in a volume that act as mount points for other volumes. Section 4 discusses namespaces and junctions in more detail.

Figure 1 shows three aggregates, containing a number of volumes, spliced together to make a single namespace.

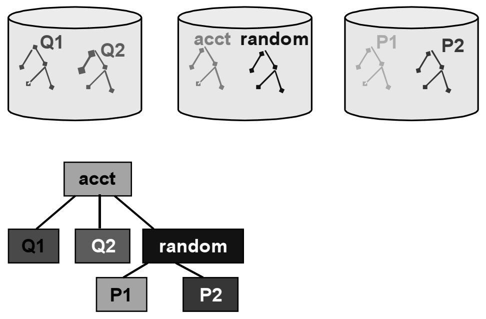

Figure 1: Example namespace

Junctions in the root volume for this namespace (*acct*) lead to volumes *Q1*, *Q2* and *random*, and junctions in random lead to volumes *P1* and *P2*.

File servers in this model, are divided into three components. Requests are received at virtual interfaces (or **VIFs**), each with its own IP address and network routing domain. These requests are initially processed by the **N,** or **networking blade**, which terminates incoming NFS and CIFS connections and maintains protocol specific state (such as CIFS connection state). The N-blade translates the incoming requests into SpinNP remote procedure calls, which are transmitted over a cluster fabric to the server responsible for the target volume. These SpinNP file system calls are in turn processed by the **D,** or **data, blade** on the target server. SpinNP file system requests can be thought of as RPC-based versions of a Vnode layer, augmented to handle the locking complexities encountered when simultaneously supporting the Microsoft CIFS protocol, NFSv3, NFSv4 and iSCSI, among other protocols. Figure 2 shows the internal structure of a two element cluster.

There are two slowly changing cluster-wide databases used to route requests and responses to the appropriate modules in the cluster. First the **volume location database** (**VLDB**) tracks both the identity of each volume s containing aggregate, as well as the D-blade that is currently responsible for that aggregate. The N-blade consults the VLDB to determine which D-blade to send a request for a particular volume. In addition, the D-blade occasionally needs to initiate callbacks to a client via a particular virtual interface. Second, the **VIF manager database** tracks which N-blade is currently hosting each virtual interface. In today s protocols, these callbacks are typically issued in support of such operations such as CIFS oplock revokes, NFSv4 delegation revokes and NLM asynchronous lock grants.

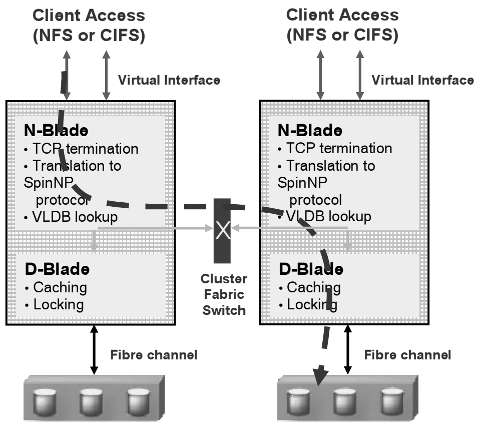

Figure 2: Example Cluster

Figure 3 shows three aggregates, one behind each GX server. Each aggregate stores two volumes which contain the junctions to produce the tree structured namespace shown on the upper right.

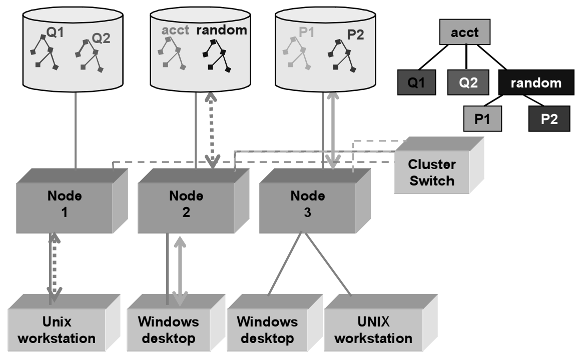

Figure 3: Overall architecture

We now walk through the details of the processing of an operation. Assume a storage client sends a CIFS request to filer node 2 with a file handle specifying a file in volume P1. The N-blade on server node 2 removes the request from the queue of the CIFS connection, and extracts the SpinNP file handle from the CIFS state and the request. The N-blade extracts a volume ID from the file handle and uses this ID to index into a cached copy of the VLDB to find the ID of the aggregate storing volume P1. Then the N-blade uses the aggregate ID to lookup the network address of the D-blade responsible for aggregate (node 3) and sends one or more SpinNP requests to that addresses. The D-blade receives the SpinNP request and executes it, sending the response back to the originating N-blade, which generates a CIFS response and sends the response back to the client. Section 5 discusses SpinNP in more detail.

3.2 Virtual Servers

The preceding section described the GX data architecture, consisting of a collection of virtual volumes distributed among a collection of aggregates owned by different servers in a cluster, and glued together to form a single tree-structured global name space. Clients typically access these volumes by contacting the server at one of several network addresses (typically IP addresses). A **virtual interface** (VIF) is a virtual network card having one or more network addresses and a corresponding routing domain, bound at any instant to a physical network card. Virtual interfaces migrate to different physical network cards in case of failures to links or servers.

This entire collection of virtual volumes accessed via a set of virtual interfaces may be virtualized as **a virtual server**. A virtual server consists of its own set of virtual volumes, with one designated as the virtual server root, acting as the root of the virtual server s private name space. A virtual server also contains its own set of virtual interfaces, and any operation received on a virtual server s interface is automatically restricted to accessing one of the virtual server s volumes. In effect, this provides GX with the capability to divide a cluster s resources into isolated sections, with each section on its own private subnet, and with users from that subnet limited to accessing data in their own section, independent of any discretionary access control lists that might exist. The alternative to virtual servers would be to have multiple clusters.

3.3 Location Independence and Single System Image

A fundamental goal of the GX architecture is the aggregation of a set of servers into a cluster that appears externally as a single large server having many volumes and many network interfaces through which those volumes may be accessed. A fundamental property of these clusters is that any volume can be accessed via any interface in the cluster, and any element can be managed via any network interface in the cluster (a well-defined exception is described in section 3.2).

Location independence provides the core underpinnings for transparent and online resource reconfiguration. Specifically, volumes can be moved dynamically between aggregates and servers in a GX cluster, and these move operations occur completely transparently to the clients, moving both data and file lock state atomically.

Similarly, the architecture allows virtual interfaces to migrate transparently between physical network cards, although today s implementation of VIF migration is not completely transparent for certain protocols: CIFS users see a TCP connection reset that may be visible as a very short lived server failure. This may be remedied by migrating the CIFS TCP connection state before doing, at least, a manual VIF failover.

The result is a clustered file system where data can be transparently moved between nodes in a cluster, users can be transparently moved between nodes in a cluster, and in general, the entire system can be reconfigured online. This level of management flexibility is required in today s environments where no suspensions of access to storage are permitted, even for adding new servers or decommissioning obsolete servers.

4 Namespace

4.1 Overview

The namespace uses ideas from the Andrew Filesystem and AFS for constructing a namespace built from a collection of storage volumes linked to each other in a tree. AFS and the GX namespace share the following properties:

- A consistent view of the namespace is provided from any client.
- Different volumes of the namespace tree can come from different server nodes on the network (termed D-blades).
- Volumes can be moved among nodes without disrupting clients or processes on the clients holding open files of the migrated volume.
- Volumes can be replicated and replicas distributed across the server nodes for purposes of load balancing and enhanced data availability.

AFS and GX diverge in that the latter does not require special purpose client software to access the namespace or to enjoy the properties described above. Instead existing file access mechanisms like CIFS and NFS can be used.

While AFS maintained its namespace via pointers to child volumes stored in the filesystem, GX uses a

junction table (maintained in the VLDB). The junction table contains **mounting relationships,** which are triples consisting of a parent volume, a child volume, and a **junction**, which is a reference to a directory-like file object that exists in the parent volume. The junction, identified by its inode and generation numbers, serves as the mount point for a child volume. A volume, identified by a **Master Data Set Identifier (MSID)**, can have multiple junctions in it, each corresponding to a mount point in the junction table. A child volume can be a parent volume to other child volumes. The junction table thus can be thought of as table of directed arcs in a graph.

Some limitations of volume mounting include:

- Only the root directory of a child volume is mounted on a parent volume's directory.
- A child volume cannot have multiple parent volumes (i.e. there may be no more than one entry in the junction table specifying this volume as a child).

The former limitation exists partly out of expediency and partly from the observation that volumes can be very small in ONTAP GX (as is the case in ONTAP-7G) [NetApp]. There's no need to mount subdirectories of a volume, when one could instead break a volume into several smaller volumes (a procedure which is simplified via the use of zero-copy volume cloning). In addition, the root of a volume is the obvious place to look for a parent volume to which to ascend.

The latter limitation exists to support traversal between a child volume and its parent volume as happens with a UNIX "cd .. issued from the root directory of a child volume. Otherwise it is not clear which parent volume to ascend to.

When an N-blade gets a request to access an object name that is a junction, it checks to see if there is a child volume mounted on the junction, instead of returning an NFS file handle that contains the MSID of the volume the junction lives on, and the fileid and generation number of the junction. The N-blade queries the junction table, using the MSID of the volume (the parent), and the inode number and generation number of the junction. If an entry is found, (and there should be at most one entry), then the entry has the MSID of the child volume, and the VLDB is queried with the MSID as a key to find the D-blade in the GX cluster that owns the child MSID.

When a D-blade gets a request to return the inode number and generation number of the parent of the root of a volume, it indicates to the N-blade that the root of the volume has already been reached. This time the N-blade needs to query the junction table using the child volume's MSID as the key. The result of the query is either the MSID of the parent volume, and the inode number and generation number of the junction the child volume is mounted on, or the result is an indication that the client is already at the root directory, of the root volume, of the namespace tree.

In the event of a D-blade failure, the ability to transit to any volume on that D-blade is lost.

The goal of the namespace is to emulate a single filesystem. However in some areas the emulation breaks down. GX does not support renaming or hard linking files across volumes. Other areas of break down are described in the next two subsections.

4.2 Considerations for CIFS Clients

CIFS supports a form of access control called Access Control Lists (ACLs). Each file can have one ACL. Each ACL contains one or more Access Control Entries (ACEs). An ACE contains a user or group identifier, and a bit mask of operations (read the file, delete the file, write the file, etc.), specifying whether a user or member of the group is to be allowed to perform the operation or to be denied the operation, and some flags. One flag is the inheritance bit. If this bit is set on a directory, then this indicates that any file or directory created inside the directory should inherit the ACE.

An issue is whether ACL inheritance should work across junctions. We considered and rejected two possibilities.

The first possibility is if the junction's parent directory has the inheritance bit set, then the ACL should be inherited in the child volume, even if the root directory of the child volume does not have inheritance set.

The second possibility is if the root directory of the child volume has the inheritance bit set (even if the parent directory of the junction of the parent volume does not have inheritance set) the ACLs should be propagated from the parent volume. The ability to dynamically reconfigure the namespace complicates these possibilities − what do we do if a volume is remounted under a directory with a different inheritance value?

In the interest of producing the least surprise, we have chosen to not inherit ACLs across junctions, regardless whether the inheritance bit is set in the parent volume or the root of the child volume.

CIFS supports a feature called "Change Notify" where a CIFS client can register an interest in being notified about changes to the directory or an entire tree of directories and files. For applications it arguably makes sense for change notify to work across junctions. However, because this has the potential to consume almost all cluster bandwidth, GX does not support change notify across junctions.

Both ONTAP-7G and GX support a feature called the security style which applies to volumes. The security style indicates whether to use Windows (ACLs), UNIX (mode bits), or a mix of the two for enforcing access. As in ONTAP-7G, the security style can be assigned per volume in GX . However if the security style is not specified when a volume is mounted, it will inherit the security style of its parent volume.

4.3 Considerations for NFS Clients

NFS clients typically run on systems that expect some conformance to UNIX semantics. One semantic used by UNIX utilities like "cp", and "mv" is to invoke the *stat()* system call on two or more files and use the returned inode number to ensure the files are different. Among other data, the *stat()* system call returns the inode number of the specified file. Suppose "m" is mount point, and "p" is a file in the parent volume, and "c" is a file in the child volume. Because the child and parent volumes are independent volumes, "p" and "c" could very well have the same inode numbers. In that case, "cp p m/c" would fail because the inode numbers of the source and target are the same. We considered and rejected using the full 64 bits of an NFS version 3 inode number to encode the MSID and inode number, because not all clients, or applications on such clients can cope with the upper 32 (or 33) bits of an inode number being non-zero. Instead, we borrow the AFS idea of computing a hashed inode number using the file's volume's MSID and the file's inode number as input. This computed inode number is used only for replies to GETATTR (get attribute) requests; the NFS file handle uses the real inode number.

We found that some NFS clients had problems when the link count of a directory did not exactly equal the number of child directories plus the entries for "." and "..". This was caused by a junction in a directory not increasing the link count. We now implement junctions as objects that behave like directories in all ways, except that permissions for search, read, and write are denied to all. This way, directory properties like increasing the link count of its parent directory are preserved.

4.4 Snapshots

WAFL snapshot semantics continue to be supported in GX with some ramifications for the global namespace [Hitz]. Supporting coordinated snapshots among volumes distributed among several D-blades would be desirable, and theoretically possible (if difficult). Snapshot coordination is currently not part of GX architecture because coordinating snapshots across all volumes of an arbitrarily-sized namespace cannot scale.

The magic ”.snapshot directory is omnipresent in GX, but because snapshots are not coordinated, the system denies clients the capability to navigate from a snapshot of a parent directory into a snapshot of a child volume. The reason is illustrated by an example. At time T a snapshot of parent volume A is taken, at T+1, child volume B is unmounted from A, and C is mounted where B was, at T+2 a snapshot of C is taken, and at T+3, the user unexpectedly encounters a snapshot of volume C instead of a snapshot of B.

5 SpinNP

The GX cluster is connected by a family of message-passing protocols called **SpinNP**. SpinNP is a layered system that defines a session and operations layer, and that specifies the requirements of its underlying transport layer. SpinNP is used for all high-traffic message passing within the cluster, both between N-blades and D-blades, as well as between different D-blades.

In developing SpinNP, we were motivated by the need for a RPC protocol with a tightly integrated session layer. We took a holistic approach to the design of SpinNP, which gave us the ability to assign function to the transport, session, and application layers as needed. We were able to build security, flow control and versioning features in from the ground up, and to design a protocol that can be run over multiple different and commonly available transports.

5.1 Transport Requirements

SpinNP sessions are layered over a network transport. Any transport can be used as long as it provides a connection-oriented protocol that provides reliable delivery of completely framed messages. We built a transport called RC, which is a simple message framing layer over UDP. The framing is required to delimit SpinNP messages on a byte stream protocol such as TCP. We also implemented other transports, including a memory-to-memory transport which is used only in the case of local communication between two blades which are co-located on the same cluster node. It is also possible to implement a transport over the PCI or other I/O buses.

Each transport provides an abstraction of one or more connections. Connections are simply pipes through which we can feed messages. Connections supply resource-limited flow control at the level of individual messages. More than one connection can be in operation simultaneously between the same pair of communicating entities.

The transport must be robust enough to inform the session layer when a connection has been lost. Timely notification of connection loss is important to speed session recovery.

5.2 Sessions

SpinNP sessions provide: strong security based on the GSS_API [Linn]; outer level flow control; the capability of exactly-once message delivery; strong major and minor versioning support; and multiple message priority channels per session.

SpinNP operations are RPC-like, and consist of one request message and a corresponding response message that is returned from the receiver to the sender of the request. SpinNP sessions support a bi-directional request flow and corresponding response flows in the opposite directions. Each SpinNP session is layered over at least three connections, one each for message priority levels low, medium and high. Low priority is used for normal request and response traffic. Medium priority is used for callback requests and responses. High priority is used for normal requests that are performed as a result of a callback. An example is the data flushes that occur after a file delegation is revoked by a callback.

The session is initially formed by the exchange of a SpinNP _CREATE_SESSION request and response. This request and response have distinguished values in their first bytes, which allow negotiation of the overall SpinNP protocol level. Once a mutually agreed-upon base protocol version has been established, the connection is authenticated by the exchange of GSS tokens. This follows the normal GSS sequence of security context creation (GSS_Init_Sec_Context), data protection and/or encryption (GSS_Wrap), and context destruction (GSS_Delete_Sec_Context). SpinNP has session level operations for each of these GSS commands. Session initialization also requires the negotiation of a set of operational protocols, called **interfaces**, which actually perform the operations that implement scalable file services within a cluster. Each interface consists of a set of operations, each defined by a request and response message pair. One of the primary interfaces is the file operations interface, which is used primarily between the N-blades and D-blades to implement network file system operations.

Each interface is separately versioned, independently of the base protocol. The base protocol version is negotiated during the exchange of the first two request and response messages. If agreement is reached on a mutually acceptable version, the session negotiation can be completed. This establishes the content of the message headers, as well as the content of a special set of requests and responses, called the session operations interface. These operations are used to perform GSS_API session initiation, and other SpinNP session negotiations.

The GSS_API is used not only to authenticate each participant in the session to the other over the initial connection, but also to authenticate each additional connection that is bound to the session. This is accomplished by the exchange of a pair of challenge-response tokens over the newly established connection. This exchange must complete before the connection can be bound to the session.

SpinNP sessions implement request level flow control. This limits the number of requests that can be outstanding in any session. The flow control combines slotted and sliding window techniques. Each session consists of a number of channels. Each channel has a sliding request window. Within a channel, requests are assigned sequence numbers. If the window size is *n,* the request with sequence number $i+n$ cannot be issued until the response has been received for all requests with sequence number less than or equal to i. This ensures that a number of requests can be in-flight at the same time, allowing a high degree of concurrency. However, the sliding window is vulnerable to a single long-running request holding up the entire channel. To reduce the likelihood of this happening, SpinNP allows the creation of multiple channels for each session. Each channel has its own sliding window, and so if one channel becomes blocked, requests can still be sent over another channel. The end-to-end windowing allows the receiver to apply back-pressure on the request channels.

There are three priority levels of channels in a session: high, medium and low. There is no direct binding of channels to particular connections, except that the following rules are applied to select a connection on which to send a message. Low priority requests and responses are restricted to the low priority channels and are sent over low priority links. Medium priority messages are sent over the medium priority channel, and can be sent over either the medium or low priority connections. Similarly, high priority messages are sent over the high priority channel, but can then be transmitted over any of the connections, high, medium or low priority.

Sessions implement an at-most-once semantic, and will not tolerate lost messages from the transport layer. Should the transport layer fail to deliver a message, the session layer can trigger session recovery, which will attempt to construct a new session to replace the failed session. It is expected that the transport will retry sending the message until it determines that it is not possible to send the message. At this point, the transport will inform the session layer that a message is undeliverable, and the session will be reset.

5.3 File Operations

The most important SpinNP interface is the File Operations interface. The requests in this interface support the CIFS and NFS network file operations. Incoming CIFS and NFS messages are translated by the N-blade to SpinNP messages, which are in a common format independent of the original source format. SpinNP file operations support all the normal CIFS and NFS file access operations, as well as locking operations.

SpinNP file operations are designed to encompass the semantics of both CIFS and NFS, including NFS versions 2, 3 and 4. It does this without relying on an indicator in the request of what the source protocol was. (Such an indicator is included in each request, but is only used for statistics gathering, quality of service monitoring, and failure diagnosis.) This requires that all expected responder behaviors be specified directly in the protocol. For example, the protocol requires that file names be accompanied by a set of flags that indicate whether case sensitive or case insensitive naming applies. Much of the challenge of implementing a server is to ensure correct operation under arbitrary source protocols without relying on the identification of the specific source protocol. An additional challenge is to select the correct behavior when the semantics of two clients, using different network file service protocols, conflict. Such conflicts are usually resolved consistently, by conventions favoring one or the other source protocol.

The file operations support all NFS and CIFS semantics, including CIFS oplocks and NFSv4 delegations. To support these features and others, a set of file callback operations is also defined. File callbacks flow from the D-blade to the N-blade, which relays them to a client, which in turn responds to the N-blade, which in turn responds to the D-blade. These callback operations are used to set the client state, for example, to recall a file or directory delegation. Such operations are necessary to ensure that the server maintains control over its resources and to give the server the ability to inform the client when it is expected to synch cached updates.

The full set of SpinNP file operations resembles NFS, and is listed below. With few exceptions (e.g. WATCH is for setting up change notify, and is specific to CIFS) NFS and CIFS requests can be handled with the same operation. The specifics of NFS or CIFS semantics are specified via flags and parameters in each request. E.g. WRITE includes a flag to support CIFS called RSRV_AT_EOF which directs the D-blade to reserve extra disk space for the file whenever a write is done that extends the size of the file.

Table 1 SpinNP File Operations

NULL	READDIR
ACCESS	READLINK
CLOSE	REMOVE
CREATE	RENAME
DISCARD_CRED	SCAN_MATCHING_LOCKS
DOWNGRADE	SCAN_FILE
FH_TO_PATH	SETATTR
FOLD_FILE	SET_CRED
GETATTR	SHARE
GETATTR_BULK	UNLINK
GET_CRED	UNSHARE
GET_PARENT	WATCH
GET_ROOTFH	WRITE
LINK	
LOCK	**call back operations**
LOCK_GRANTED_ACK	NULL
RECLAIM_LOCKS_ACK	LOCK
LOOKUP	MOVE_LOCKS
MOVE_LOCKS	NOTIFY
OPEN	RECALL_LOCK
PREFETCH	RECLAIM_LOCK
READ	RETRY

While we originally planned to build a replay cache into the SpinNP session layer, we discovered we could achieve better end-to-end resiliency by building a replay cache in the SpinNP file operations layer, and that this can also provide the functionality required in an NFS response cache. It proved to be simpler and more effective to provide a single end-to-end solution than to chain together multiple links protected by different replay caches.

5.4 SpinNP IDL

We created a new interface description language (IDL) for SpinNP. This IDL has features that facilitate direct compilation of marshalling and unmarshalling code from the SpinNP specifications. The IDL relies heavily on two variable structures. The first is the **switch**, which selects variable fields from one of several alternatives. The switch is comparable to a C union; however, the selection of a branch is explicitly determined by a discriminator which is the first element of the switch construct. The total size required by a switch is determined by the selected branch, not by the size of the largest possible branch. The other is the **collection**, which can select any number of variable fields, specifying which fields are selected by a bitmap specification which appears at the beginning of the collection.

Examples of a switch and a collection from the SpinNP file operations specification are:

```
struct fileop_req_msg_body_t {
  src_protocol_t            src_protocol;
  fileop_request_flags_t    fileop_flags;
  cred_t                    cred;
  switch (fileop_proc_num_t proc) {
     case NULL_FILEOP:
       null_fileop_req_t     null_fileop;
     case ACCESS:
       access_req_t   access;
       …
  } s;
};

collection file_attr_t
     (file_attr_types_t fa_type) {
  case FA_CHANGE:
    uint64_t          fa_change;
  case FA_SIZE:
    uint64_t          fa_size;
  …
};
```

The SpinNP IDL is strongly typed, which facilitates compilation of code directly from the specification. We developed an IDL compiler to build C, C++, Perl and Ethereal code all from the same input file, which is the written specification of the interface. Having an exact match between the protocol specification and the code, including Ethereal descriptions, significantly sped development [Lamping].

One of the most important features of the SpinNP IDL is its strong support for versioning. It is possible to annotate a single copy of a SpinNP interface specification document so that multiple versions of the protocol can be specified by the same document. This gives a clear indication of the development of the protocol, and ties directly into the interface compiler, which generates marshalling and unmarshalling code that is aware of the differing contents of each version.

6 System Management

System management in the ONTAP GX system is responsible for maintaining the illusion that a GX cluster is a single system, even as new servers join the cluster, old servers are removed from the cluster, data is migrated between aggregates, and network addresses migrate between physical network cards.

Management is also responsible for ensuring that a large cluster hides the failures of internal components, so as to give the appearance not only of a reconfigurable cluster, but a cluster that never stops providing service.

The rest of this section describes the details of GX system management.

6.1 Replicated Database

The heart of the system management software is the replicated database, RDB. While RDB (not to be confused with Oracle s RDB) is mostly a new design, it uses some ideas from the "ubik" library used by AFS.

Databases built atop RDB store all cluster wide configuration information, including the location of all the cluster s virtual and physical resources, and the state of any long-running tasks that need to survive the failure of any given node. Each replica of an RDB database is guaranteed to be equivalent.

RDB provides two significant types of functionality.

First, it provides an election mechanism, used to elect a coordinator for updates to the underlying database. All updates to the database are funneled through this elected master, allowing a relatively straightforward implementation of the database. In addition, the elected master can perform application specific tasks that require at most a single instance to run successfully. For example, the elected master for the virtual interface manager also runs the single instance of the task that moves virtual interfaces between physical network cards in the event of a link or node failure in the cluster.

Because a new election is performed after a node failure, RDB can keep a database application running in the face of multiple node failures.

Second, RDB provides a transactional database facility allowing structured records to be atomically added to any system management databases. RDB databases are replicated among a set of machines in the cluster (referred to as a ”ring). Currently, all machines in the cluster are in the RDB ring, and updates can be performed to the database as long as a majority of the servers in the ring are operational. A proper majority is required to unambiguously resolve the case where a network partitions. As RDB transactions are created, they contact the master for an (epoch, transaction ID) pair. The transaction ID is essentially a transaction count since the last time the cluster changed masters. The epoch is a sequence number of the number of times the master changed. Thus, by sorting on epoch and then ID, we get a global ordering of transactions in a cluster.

GX and RDB do not guarantee that the state of a database will be the same for the duration of an operation on a cluster. For example if a volume is one D-blade when a CIFS request arrives to file on the volume, the volume could move when the N-blade issues a SPINNP request. The N-blade would get an error indicating the volume has moved, and re-consult RDB for the volume s new D-blade.

A number of the databases built on top of RDB are discussed below.

6.2 Volume Location Database

The volume location database (VLDB) is an RDB replicated database. The VLDB stores a number of tables used by N-blades and the system management processes to locate in several steps - the D-blade corresponding to a particular volume. First, the volume is mapped to the aggregate that holds the volume. Second, because ownership of entire aggregates can be passed from one system to another, as in the case of system failure, the aggregate ID is mapped to a D-blade ID. Third, the D-blade ID is mapped to its network addresses via another RDB-based database, part of the VIF manager, described below. In addition, if a junction is encountered, the N-blade consults the junction table (kept in RDB, as described in section 4), to find the mounted child volume. Note that all of these mappings are composed and cached in the N-blade s memory, so that the results of all four lookups are typically available after a single hash table lookup.

The contents of each of these tables change during certain operations. When a volume is moved between servers, or between aggregates on the same server, the volume is first moved, and at the very end, the VLDB is updated to indicate that the volume has a new home aggregate. If a server in a storage failover partnership (where two D-blades share access to same set of disks and can takeover from each other in event of failure) fails, the surviving server updates the VLDB s aggregate to D-blade mapping, so that all of the N-blades in the system know where to find the aggregate. And if a system administrator changes the network address of an N-blade or D-blade, the blade s cluster IP address set is updated in the VIF manager s database as well.

6.3 Virtual Interface Location Database

The Virtual Interface (VIF) location database is managed by an active RDB process called the Virtual Interface manager (the VIF manager). The VIF manager is responsible for tracking which virtual network interfaces are associated with which physical network cards in the cluster, and ensuring that, should a server fail, that the IP addresses associated with the failed VIF are moved to an acceptable backup network card.

It is obviously important − regardless of what type of failures occur − that a given VIF is exported by no more than one physical network card at any instant. To provide this guarantee, VIFs are divided into two classes, fixed VIFs, which never migrate to other systems, even upon link or host failure, and moveable VIFs, which can be moved between servers in a cluster. If a cluster loses more than half of its servers, the RDB database will be unable to elect a master, and every VIF manager will see, after a small timeout, that it is no longer in contact with the RDB master for this database. When this occurs, all moveable VIFs are torn down, since the VIF manager can not distinguish between a majority of the servers in a cluster being down, and a network partition, where the master VIF manager process is simply no longer reachable from the local VIF manager. In the latter case, however, the VIF manager will likely reassign the moveable VIFs to another system, and to avoid having a cluster with duplicate IP addresses from different ports, a VIF manager that can t contact the VIF manager RDB master must tear down its moveable VIFs. Because fixed VIFs can never migrate to other servers, these VIFs can continue operation even after loss of contact with the RDB master, and indeed, this is the sole reason for the distinction. Of course, it is very rare for a cluster to lose more than half of its servers; typical failure modes are single server failures and network partitions, where a majority of servers remain in contact with each other.

6.4 System Management Framework

The system management framework provides administrative interfaces to the cluster. It is based upon the eponymously named SMF from Marconi Corporation. The system management framework generates a web user interface, command line interface and SNMP tables based upon a set of tables provided as input.

The commands invoked by this framework update configuration state stored in RDB, and thus mirrored to all servers in the cluster.

6.5 Job Controller

The job controller module is a part of the system management framework that provides support for long running operations that need to be restarted or cleaned up after a server failure.

The job controller provides RDB-based stable storage to keep track of the progress of a job. For example, a volume move operation is a complex multi-step operation with significant temporary state that must be cleaned up in the event of an error. A volume move begins by creating a volume on the destination server and creating a snapshot at the source server. The contents of the snapshot are then propagated to the empty volume at the destination. The snapshot propagation cycle is repeated several times and finally the VLDB is updated to point to the volume s new location. With this implementation, it is clear that if a crash occurs during the move, significant clean up must be performed before the operation can be restarted. The job controller provides a straightforward mechanism to ensure that the cleanup operations are invoked and the job restarted.

7 False Starts

The original design of junctions included the MSID of the child volume. This would have complicated data protection logic. For example, a volume might be backed up, and then the volume or subset thereof restored in a different place in namespace. Without additional logic, the restored volume would contain junctions pointing to child volumes that existed in other parts of the namespace.

The SPINNP protocol conveys semantics in a file access protocol independent manner. In most cases, the cost of this generality is nominal. However, when the D-blade converted READDIR results from WAFL to SPINNP, and the N-blade converted SPINNP READDIR results to NFS READDIR results, the CPU overhead was significant. We extended SPINNP to support NFS formatted READDIR results, and gained about 5% throughput using the benchmark described in the next section.

8 Performance

8.1 Overview

The system's performance characteristics derive from the division of the file service process into separate protocol termination and disk service modules (the N-blade and D-blade modules, respectively). As described in section 3.1, the high performance path consists of client requests traversing an N-blade routing across the cluster network and terminating at the appropriate D-blade (as illustrated in Figure 2).

The resources that drive performance include the clients networks, N-blade CPU and memory bandwidth, cluster network, D-blade CPU and memory bandwidth, and disk bandwidth. For typical NAS access patterns, we find the CPU balances across the N- and D-blades. A zero-copy networking stack removes the memory bandwidth bottleneck. Disk subsystem delays are ameliorated by adequate memory cache, including NVRAM to minimize latency of writes, on the D-blade.

8.2 Scaling

From a data traffic perspective, each N-blade is acting as no more than a switch to one of several other D-blades, which simplifies the analysis of the scaling limits of GX.

The performance of a node in a cluster (in operations per second) will be:

$$\begin{aligned} &performance\_of\_local\_operations \\ &+ performance\_of\_remote\_operations \end{aligned}$$

Where **performance_of_local_operations** is the performance of operations received by an N-blade that go to the D-blade that exists on the same node, and **performance_of_remote_operations** is the performance of operations received by the N-blade that go to D-blades that are not one the same node as the N-blade.

Assuming each N-blade receives an equal share of load from external clients, and assuming each N-blade evenly switches its received load to each D-blade, then among the D-blades, each D-blade is processing the same load. Let **P** be the performance of a single node cluster. Then in an **N** node cluster $performance\_of\_local\_operations = P / N$,
and $performance\_of\_remote\_operations = P \times (N-1) / N \times E$,
where **E** is the performance efficiency of remote operations. Thus the expected performance of a cluster of N nodes is:

$$\begin{aligned} &\text{expected\_performance\_cluster}(N) \\ &= N \times \left(\left(\frac{P}{N} \right) + \left(P \times \frac{N\text{-}1}{N} \times E \right) \right) = P + P \times (N-1) \times E \\ &= P \times (1 + (N-1) \times E) \end{aligned}$$

(For N > 0)

8.2.1 Scaling Results

To measure scaling, we used the SPEC SFS91_R1 V3.0 benchmark[*] to generate a standard workload of NFS (version 3) operations/second [Capps]. We present SFS numbers, because the SFS workload has a mix of I/O and metadata operations (both modifying and non-modifying) and is derived from some real customer workloads. SFS has uniform access rules that require no partitioning of data among the devices that comprise the system under test. Thus the SFS matches the assumptions listed in the previous section.

We rarely run large cluster performance benchmarks because of the labor and capital costs and because we ve found that setting up a two-node cluster, and configuring the benchmark to direct 100% of each N-blade s SpinNP traffic to the other (remote) D-blade is a sufficient predictor of how the large cluster will scale. As a result, we have just two large cluster figures to offer.

On a single mid range cluster node we achieved about 20,900 operations/second. On the two node ”100% remote cluster we achieved about 17,900 operations/second. Note that neither of these runs were compliant from SPEC s run rules, because the goal was to determine the maximum throughput, not to produce a compliant run. Thus, fewer than the required 10 ”load points were attempted with each single node run (see page 55 of [SPEC]). From the single node and ”100% remote runs we derive an expected efficiency of 85.6%. A 20 node cluster achieved about 318,000 operations/second versus an **expected_performance_cluster(20)** value of 360,000 operations/second. We did not have the same number of disk drives per node in the 20 node cluster as in the one and two node cluster, and believe that accounts for the discrepancy. We did not re-run with more drives because we were aiming for a much higher number.

Using our ”high end cluster node, a 24 node cluster configuration measured 1,032,461 SPECsfs97_R1.v3 operations per second, (with a corresponding overall response time of 1.53 milliseconds; for a definition of overall response time, see pages 55-56 of [SPEC]). Each node had three one gigabit/second Ethernet controllers, one for handling client traffic, two for the cluster interconnect. We do not have single node and 100% remote two cluster node numbers for the same software version on the high end node. However, several months after the 24 node run was published at SPEC s web site, we measured the high end single node at about 55,000 operations/second, and each node of the 100% remote two-node cluster at about 41,000 operations/second per node. The expected efficiency is about 75%, and **expected_performance_cluster(24)** = 1,003,750 or below 2.8% of actual results.

8.3 Load Balancing

In any collection of systems intended to service a common pool of clients, balancing load across the collection is an important consideration. We discuss two techniques used to address the problem.

8.3.1 Rebalancing Load

A fallacy of the previous section is the assumption that loads to the cluster or inside the cluster will be balanced. Because hot spots are a reality, GX offers two axes for balancing load.

The first axis is the ability to transparently migrate volumes among D-blades. With this capability, a given D-blade, and indeed a given set of disks, need not be a hot spot. Since volumes can be made arbitrarily small (while still being dynamically expandable), then, subject to the maximum file size needs of the application, one can create many small volumes that can be independently moved around the cluster as frequently as needed. The smaller the average volume size, the faster it can be moved in reaction to a load

[*] SPEC ™ and the benchmark name SPECsfs97_R1 ™ are registered trademarks of the Standard Performance Evaluation Corporation. For the latest SPECsfs97_R1 benchmark results visit www.spec.org (or more specifically: www.spec.org/osg/sfs97_R1).

imbalance and the easier to find a D-blade with sufficient spare cycles to service it.

The second axis is the ability to transparently migrate virtual network interfaces (VIFs) from one N-blade to another. The more VIFs a cluster has, the easier it is balance load across N-blades. Each N-blade has multiple VIFs, perhaps even multiple VIFs that refer to the same namespace. As the number of VIFs approaches the number of external clients, or the even the number of unique client source network addresses (for clients with multiple interfaces), the flexibility to respond to clients that dominate the capacity of a single N-blade improves. The clients that are using a VIF on an N-blade can be migrated by moving the VIF to an N-blade that has less load.

8.3.2 Load Balancing Mirrors

The root volume of a namespace, if not mirrored, becomes a performance bottleneck even if the traffic is mostly LOOKUP operations. To prevent the D-blade that holds the root volume of the namespace from being a bottleneck, a best practice for GX is to have a read-only load balancing (LB) mirror of the root volume of the namespace on each D-blade of the cluster. Each mirror is created via an asynchronous volume SnapMirror operation, which is similar to the asynchronous VSM feature in ONTAP-7G [Patterson]. This way, any read-only external client request (CIFS, NFS, etc.) that accesses the root volume can be serviced from any D-blade. Since external client requests arrive at N-blades, and since each N-blade will tend to have a local D-blade on the same node, the external request can be serviced locally, rather than being serviced by a D-blade that is on another node.

The set of load balancing mirrors, plus the writeable master volume, is collectively called a **volume family**. Each member of a volume family has a unique data set identifier (DSID), but each member shares the same MSID. This way, NFS requests can be routed to any available load balancing mirror for an MSID that appears in the request's NFS filehandle.

If a volume is a load balancing mirror, clients are permitted to access and modify the writeable master, but only if the access path is prefixed by the component "/.admin". Note that whether a volume in the path has a load balancing mirror is immaterial; the writeable version of the volume is always accessed. The ".admin" component is not quite a magic directory like WAFL's ".snapshot":

- It only appears at the root of the namespace.
- For NFS, only mount operations see it; NFS file and lock operations do not.

Thus if an NFS client mounts "/", and then tries to "cd" to ".admin" the LOOKUP will fail. Whereas, if the NFS client mounts a path starting with "/.admin", all access will be via the writeable masters for each volume family. GX accomplishes this by using one bit in the NFS file handle to indicate whether the writeable tree prefixed by "/.admin" is to be used or not.

Even though ".admin" appears at the root of the namespace this does not mean only the root volume in a namespace can have load balancing mirrors. Any volume can have load balancing mirrors. The writeable master of a volume family is always reachable by a path that starts with "/.admin".

If a writeable master is modified, a client accessing the volume through the normal path will not see the update until the system administrator directs the propagation of the updates to the load balancing mirrors.

8.4 Cluster Expansion

Ultimately, even with a perfectly balanced load, the aggregate load can exceed the cluster's capabilities. GX allows one to add nodes to the cluster, and thus provide new, idle places, to which to migrate load.

Nonetheless, expansion and migration will not suffice for all types of loads. A single volume can potentially experience more demand than the load capability of a single D-blade. A future version of GX will allow volumes to stripe across D-blades.

The problem of a single client outstripping an N-blade, or an N-blade's physical network interfaces is harder to solve, and ultimately requires external clients both capable of network trunking, and knowing the striping in the case of striped volumes.

9 Experiences with GX

We summarize the experiences of three customers who use GX today.

The first customer a provider of computer generated special effects for motion pictures - found that GX and its predecessor from Spinnaker provided the only storage solution capable of supporting its rendering load. Via the transparent volume move feature, the customer rebalanced storage usage across its GX nodes, and achieved average storage capacity utilization of nearly 90%, adding storage as needed.

The second customer is a supercomputing center supporting the information technology needs of scientists and other academic researchers. The customer uses a six node GX cluster for storing results of workloads and for user home directories with data consisting of mainly small files. The cluster handles bursts of data of up to 600 megabytes/second of mostly NFS traffic with some CIFS.

The third customer is a semiconductor manufacturer with a four node cluster. A single attached client can achieve 250 megabytes/second reading or writing, and with a 75/25 % mix of read and write, a single client achieves 650 megabytes/second. From 224 clients, an aggregate 800 megabytes/second write speed was measured, and aggregate gigabyte/second read (direct I/O) speed was seen, and using the 75/25 % mix 1.2 gigabytes/second read/write speed was achieved.

10 Conclusions

Data ONTAP GX provides many of the best features of previous scalable namespace file servers. It does this through widely deployed network file system protocols, including NFS and CIFS, avoiding changes to client software. The architecture translates file access requests to a common backend protocol that is used for all high-traffic communication within the cluster. This simplifies the implementation of the backend data server modules. The entire cluster is administered through a single interface, and administration is implemented through a number of cluster services. Performance scalability is linear, achieving a rate of over one million NFS operations per second on a 24 node cluster.

11 Acknowledgements

We thank Rich Sanzi, Darren Sawyer, Margo Seltzer, and five anonymous reviewers (provided by USENIX) for their constructive critiques of several drafts of this paper; and Dennis Chapman and Tianyu Jiang for information on customer experiences.

12 References

[Anderson] Anderson, D. et al, ”Interposed Request Routing for Scalable Network Storage ACM Transactions on Computer Systems, vol 20, no 1, Feb 2002

[Campbell] Campbell, R., ”Managing AFS: Andrew File System, ISBN 0-13-802729-3, 1998.

[Capps] Capps, D., ”What s new in SFS 3.0 , NFS Conference, 2001.

[Hitz] Hitz, D. et al, ”File System Design for an NFS File Server Appliance, USENIX Conference Proceedings, 1994.

[Howard] Howard, J. et al, ”Scale and Performance in a Distributed File System, ACM Transactions on Computer Systems, Vol. 6, No. 1, 1988.

[Kazar1990] Kazar, M. et al, ”DEcorum File System Architectural Overview, USENIX Conference Proceedings, 1990.

[Kazar2002] Kazar, M., ”High Performance and Distributed NAS Server Architecture for Scalable and Global NFS file systems, NFS Industry Conference, 2002.

[Kleiman] Kleiman, S., "Vnodes: An Architecture for Multiple File System Types in UNIX," Proceedings of the USENIX Conference, 1986.

[Lamping] Lamping, U. et al, ”Ethereal User s Guide, http://www.ethereal.com, 2005.

[Linn] Linn, J., ”Generic Security Service Application Program Interface Version 2, Update 1 , RFC 2743, Internet Engineering Task Force, 2000.

[NetApp] Network Appliance, Inc., ”Introduction to Data ONTAP™ 7G, TR 3356, 2005.

[Neuman] Neuman C., ”The Kerberos Network Authentication Service (V5) , RFC 4120, Internet Engineering Task Force, 2005.

[Patterson] Patterson, H. et al, ”SnapMirror®: File System Based Asynchronous Mirroring for Disaster Recovery, USENIX Conference on File and Storage Technologies Proceedings, 2002.

[SPEC] Standard Performance Evaluation Corporation, ”SFS 3.0 Documentation Version 1.1, 2001.

[SNIA] ”Common Internet File System Technical Reference, Storage Networking Industry Association, 2002.

[Sun] Sun Microsystems, ”NFS: Network File System Protocol Specification, RFC 1094, Internet Engineering Task Force, 1989.

[Thekkath] Thekkath, C. et al, ”Frangipani: a scalable distributed file system, Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles, 1997.