UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Overcoming write penalties of conflicting client operations in distributed storage systems

Master of Science

at the Paderborn Center for Parallel Computing
Department of Computer Science
University of Paderborn

Markus Mäsker
Paderborn, September, 11, 2012

Supervisor:
Prof. Dr. André Brinkmann
Prof. Dr. Stefan Böttcher

Paderborn
Center for
Parallel
Computing

Erklärung / Declaration

Abstract

Reliability is a crucial aspect of distributed file systems (DFS) as failing devices are the rule rather than the exception. The pNFS standard permits the use of RAID algorithms to generate data redundancy and therefore, recover from data server failures. Unlike hard disk RAID systems, DFS lack a central RAID controller. The metadata servers of current distributed file systems coordinate client requests and try to resolve data server failures. In my master thesis I introduce a coordinated data server architecture that allows the data server cluster to perform access management and provide consistency control on a peer-to-peer basis. The architecture provides a higher degree of parallelism than current DFS while preserving the consistency and reliability of a local RAID controller.

Contents

1. Introduction

The total amount of data stored worldwide exceeded the zettabyte barrier in 2010. A recent Symantec survey[1] revealed that this amount grew to about 2.2 zettabyte in June 2012. Small to medium sized companies on average store 563 terabytes with an annual growth rate of 178%. Enterprises store 100.000 terabytes and have an annual growth rate of 67%. Therefore, the demand for large storage systems is big and keeps getting bigger. Peter Ungaro, President and CEO of Cray, states in his keynote[29] of the 28^{th} IEEE (MSST 2012) that Moore's law[19] does not apply to disk-drive performance. While the CPU power is increasing by a factor of 16 over a 10 year period, the disk's speed only increased by a factor of 8. Therefore, the great challenge that comes along with the increasing demand is performance rather than capacity.

In order to handle the gap between processing and disk I/O performance, multiple disks are used in parallel. Research in the parallel usage of multiple disks started with the introduction of Redundant Array of Inexpensive Disks (RAID)[21] in 1988. Several disks are combined by a special controller and form a large logical disk. Data written to the logical disk is split into small chunks and scattered over the individual disks. In consequence, the RAID system not only provides the combined capacity as a logical address space, but also provides the combined bandwidth of the disks for individual read and write operations. The downside to this is the increased risk of data loss. Should any of the disks fail, all data is corrupted. Therefore, RAID systems store additional, redundant data. Consider a RAID controller with four disks attached forming a RAID 5 configuration. Incoming data is split into three stripe units SU_0, SU_1 and SU_2. The redundant data is calculated as follows: $SU_p = SU_0 \otimes SU_1 \otimes SU_2$. These four units build a stripe and are scattered over the four disks. Should any of the disks fail only one of these blocks is lost, for example SU_0. Once the failed disk has been replaced with a new one, the RAID controller start to rebuild the stripe: $SU_0 = SU_1 \otimes SU_2 \otimes SU_p$. The disk failure did not lead to data loss. However, this imposes an issue called small write problem[26]. Every time one of the stripe units is updated, the RAID controller also has to update the parity block SU_p in order to maintain the stripe's consistency.

With the development of distributed file systems, the RAID technique has been adopted to the server level. These server clusters allow the creation of large file systems with thousands of disks and being able to provide access for thousands of clients. January 2010, after several years of development, the Internet Engineering Task Force (IETF) published a standard[22] adding parallel data server access to the NFS protocol. However, spreading data of a file over several data servers might improve the read and write performance, but also increases the risk of data loss. Should any of the server fail, the file is inaccessible. A study[30] at Microsoft's server farms showed that per 100 servers about 16 hardware failures occur every year. Therefore, redundancy, such as RAID 5, needs to be added to the data server cluster as well. At the cluster level the same small write problem exists, but handling it is more complicated. The RAID controller can guarantee the atomic operation of the read-modify-write process of the stripe units. In contrast to that, the data server cluster does not have a central controller. The pNFS storage protocol RFC5664[3] advises the metadata server to serialize the write operation. A possible way to achieve this is a MDS managed stripe-lock. The client updating a stripe unit has to acquire the stripe-lock, then read the old parity unit and calculate the new one. Both the parity unit and the stripe unit are written to the appropriate data server and the stripe-lock is returned to the MDS. A second client that wants to update another part of the file, but happens to share the same parity unit, has to wait until the lock is returned. This workflow not only limits the throughput for small write operations, but also moves stripe consistency responsibility to the client and increases the load of the metadata server. The client has to monitor the state of the write operation itself and, in case of an error, try to recover the stripe consistency or report the error to the metadata server. The MDS then has to restore a consistent state.

In this thesis I will design a peer-to-peer based data server cluster. The idea is based on the TickerTAIP Parallel RAID Architecture[5] introduced in 1994. It describes a parallel architecture of loosely coupled hosts that coordinate the functions otherwise performed by a RAID controller. Thereby it provides fault-tolerance, scalability and atomic operations. This 18 year old approach is applied to the data server of the parallel file system architecture pNFS. The data server cluster coordinates client write operations internally and provides a lock-less, single-parity consistency model. Instead of a central instance, such as the metadata server, the data servers coordinate the write operation and guarantee stripe consistency. The client only has to send the new data to the appropriate data server. This then calculates the difference of the existing and the new data and forwards this to the server responsible for the parity unit of the stripe. This server updates the parity unit accordingly. I will present a workflow that provides atomicity of the described operation and handles error cases. The consistency is maintained using a global state and version management model.

The implementation will cover both the data server, the according client library and the necessary network protocols. The evaluation will compare the performance of conflicting client write operation both for the coordinated cluster workflow and a MDS managed stripe-lock mode. The tested configuration will cover nine data server, with one of them managing the parity unit. The evaluation will show that the cluster-internal coordination provides a higher degree of parallelism for conflicting client operations and the eight stripe units are able to saturate the parity server's hardware to over 90 %. Furthermore, the latency of the workflow modes is measured and compared. We will see that the driving factor of an operation's latency is the network bandwidth. Even for non-conflicting operations the latency of the stripe-lock mode is slightly higher than the latency of the coordinated cluster mode.

Finally, this thesis concludes that the coordinated cluster architecture provides a cleaner separation of concerns between the metadata server, client library and data server. It is competitive with an externally managed access control workflow, such as the stripe-lock approach, at non-conflicting client operations and outperforms the stripe-lock mode for conflicting client operations.

2. Related Work

As described in the introduction, the system is inspired by the TickerTAIP Parallel RAID Architecture[5]. It describes a set of distributed processes that coordinate client requests in a way a centralized RAID controller would. However, the architecture does not provide serializability of conflicting write requests. Furthermore, the algorithms are designed under the premise that the parity calculation is the limiting factor of the system. This was a valid assumption in the early 90^{th}, but does not apply to modern symmetric multiprocessing (SMP) architectures.

My cluster used Panasas' object-RAID[11] mechanism to stripe data across the data server cluster. Object-RAID is used in PanFS, which supports striping of files with RAID-5 redundancy mechanics. On the one hand, it uses client-driven parity calculation, which reduces data server load[33]. On the other hand, this requires the client to monitor the operation and recover from failure. The metadata manager needs to scrub parity, if the client fails. The client can read the parity unit and perform an end-to-end integrity check based on the parity information. Furthermore, the per-file RAID protection allows the reconstruction of multiple files in parallel. Every node can be used to reconstruct lost data and the per-file RAID limits the effect of failed components to individual files. In contrast to PanFS, my cluster is will handle the parity calculation of write operations and manage recovery operations internally.

The parallel cluster file system PVFS[6] also stripes data across several data servers, but does not support parity-based redundancy. Client operations are ordered in a way that provides atomicity and allows both stateless clients and server[20]. Therefore, no lock management is necessary. In Lustre server level striping with redundancy support is called Server Network Striping (SNS)[28], but has not been released yet. Lustre stores data in form of files at the host's file system. Lustre's distributed locks are managed by the Object Storage Target (OST) responsible for the requested data. In contrast to Lustre, which makes use of redundant hardware to reduce server failures, Ceph[31] creates replicas of data objects reducing the effect of failing nodes. Similar to my architecture, Ceph's object storage daemons coordinate write operations and maintain replica consistency[32]. A client writes to the first replica and that node then manages the update of the other replicas. Ceph is one of the few distributed file systems that have direct coordination among the storage nodes. GoogleFS[10] uses a similar mechanism. It is developed to fit Google's application requirements best and therefore, focuses on write once, read many workflows. Each data chunk is written to a primary node and several replica nodes. The data flow also is cluster internal, but highly adapted to the network architecture in order to provide high bandwidth utilization. The write operation, including the replica distribution, is coordinated by the primary node. GoogleFS also uses server managed locking to serialize write operations. IBMs cluster file system GPFS[24] uses a distributed locking mechanism and uses replication techniques, not parity redundancy.

The pNFS protocols[22][23][4][3] have been published in January 2010. Whereas the client library has been added to the linux kernel, a pNFS server implementation has not exceeded the prototype level, yet. None of the above file systems provide pNFS support and neither will my architecture.

K. Amiri, G. Gibson and R. Golding[2] compare several concurrency control protocols. First, they introduce two centralized locking protocols. One using a dedicated lock server and an immediate lock return by the client, the other allows caching of the lock at the client side for a defined lease time and a callback that the server issues to relinquish a cached lock. In order to overcome the potential bottleneck of a centralized lock server two distributed lock schemes are presented. One uses parallel lock servers, which introduce the potential of a deadlock or increase latency due to deadlock avoidance. The other protocol works highly parallel by making each storage node the lock manager of its own data. In order to reduce the messaging overhead, lock acquire messages can be integrated into I/O requests. The fifth protocol uses a lock-less approach with the help of timestamp ordering. Each operation is defined by base storage transactions (BST) which are tagged with a unique timestamp. Based on this timestamp conflicting operation are ordered and performed atomically. The paper shows that device-served lock management is a feasible approach in order to build scalable, highly concurrent, shared storage. These protocols still favor a client-driven read-modify-write workflow, whereas my storage cluster both handles the operation coordination and parity block updates.

Transaction processing is a crucial part of my coordinated cluster. A very popular mechanism for distributed transaction management is the two-phase commit protocol (2PC)[12]. While this protocol works fine in the absence of failure, a failing node leads to a blocking state. In order to overcome this issue, the three-phase commit protocol (3PC)[25] provides a non-blocking workflow. The Paxos[16] protocol family provides several consensus algorithms with varying messaging overhead. The Paxos algorithms can include leader election mechanisms and tolerate a predefined number of failed nodes. Gray and Lamport[13] compared the two-phase commit protocol and the Paxos commit protocols and concluded that the Paxos commit protocol family has the same message latency in a fault free case and provides higher fault tolerance. On the down side it produces more messages than the 2PC. My workflow is based on the two-phase commit protocol. The stripe's parity server always acts as the coordinator, therefore, there is no need for leader election. In order to compensate failing nodes, each operation is monitored for a timeout and consensus is built on either commit or abort of the operation. Should the parity server, and in consequence the coordinator, fail, the other participants detect a timeout and agree on a new coordinator. The new coordinator is defined deterministically by the client operation's attributes and does not need a voting mechanism.

3. Basics

In this section I will provide the reader with a fundamental understanding of the terms and technology used in this thesis.

3.1. Distributed File Systems

The term *distributed file system* is used to describe a file system that stores data at multiple servers. There are several specializations such as distributed fault-tolerant file system. These systems provide fault-tolerance by adding redundancy to the stored data and lack a single point of failure. Distributed parallel file systems split files into small chunks and distribute these across multiple servers. This allows clients to read and write data of a single file from several servers and therefore, achieve high-performance I/O. A combination of both approaches is a distributed parallel fault-tolerant file system. These provide parallel access and fault-tolerance.

3.1.1. PNFS Architecture

With the Network File System (NFS) Version 4 Minor Version 1 Protocol[22] parallelism was introduces to the NFS standard.

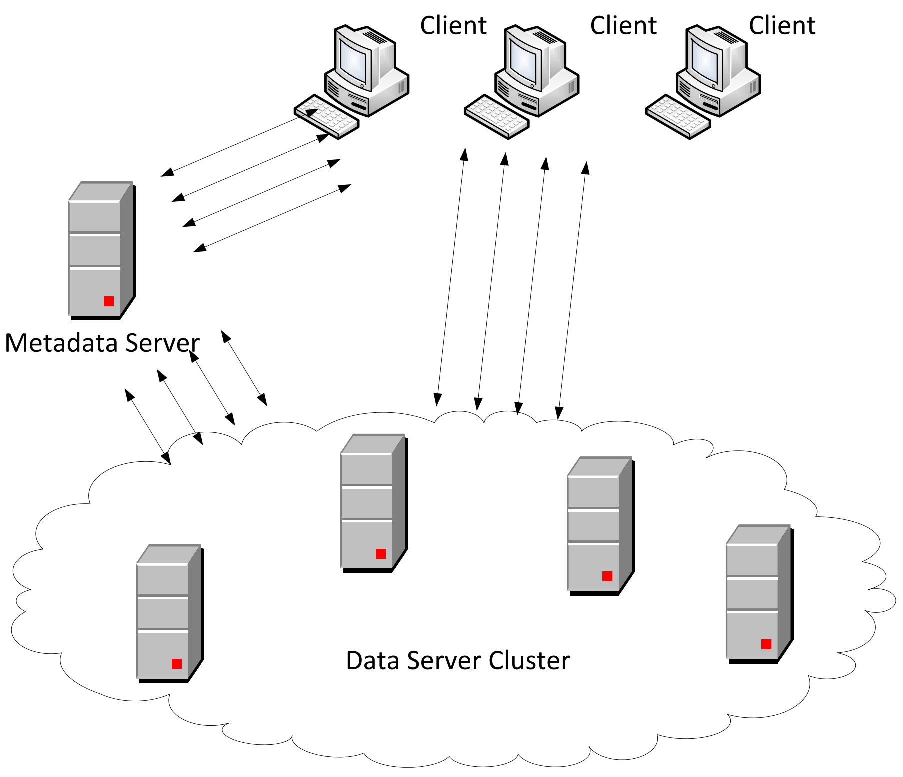

Fig. 1: pNFS system architecture

Figure 1 shows the architecture of a pNFS environment. It consists of one metadata server (MDS), many data servers (DS) and lots of clients. The standard itself only defines the use of one metadata server. Modern large scale distributed file systems, such as Lustre[1] or PanFS [2], use multiple MDS in order to provide fault tolerance and serve thousands of clients. There has been a draft titled *Metadata Striping for pNFS*[7] which discusses the support of clustered metadata servers in pNFS. However, it expired April 21, 2011 and there does not seem to be further development in this area.

The metadata server is responsible for the file system's management tasks. This includes the POSIX file system operations, such as create, update, delete or readdir. Further features are access control to files, consistency control of redundant data, recovery of failed devices and client session management. Issues regarding the consistency control are discussed in section 3.2.2 in more detail.

The data server (DS) store the actual user data. There are three protocols that define the data transfer between the clients and data server. A block-based approach using fiber channel or iSCSI connections[4], a file-based using normal NFS access[22] and an object-based. This Object-Based Parallel NFS (pNFS) Operations[3] treats chunks of data as objects and provides RAID capabilities. The RAID technology will be handled in section 3.2 in more detail. With this technique data is striped across several data servers in parallel and the I/O bandwidth of the individual servers can be combined to provide higher throughput.

The client performs file system operations on the MDS and I/O operations with the data servers. Depending on the used storage protocol, parallel access to the data server can increase reliability and throughput. Fedora 15 was the first Linux distribution with pNFS client support.

3.2. RAID Technology

The RAID technology was introduced by Patterson, Gibson and Katz[21] in 1988. Multiple physical disk drives are connected to a RAID controller and merged into a single logical drive. The controller splits incoming data into small chunks, called stripe unit (SU), and stores these at the individual disks. The distribution of the chunks to the disks is defined by the used RAID level.

3.2.1. RAID Level

There are too many RAID strategies to discuss them all. For this thesis it is sufficient to discuss the fundamental concepts. At *level 0 (striping)* all chunks are distributed in a round robin fashion to the disks. The I/O bandwidth of all disks is used in parallel. This provides a high performance, but increases the risk of data loss. Since no redundancy is created a single disk loss corrupts the complete disk array.

At *RAID-1 (mirroring)* two disks form a group. Every SU stored on one disk is also cloned to the other. A single disk failure can be compensated

[1]lustre.org
[2]panasas.com

by the mirrored disk and does not imply data loss. The read performance is comparable to level 0, because two different read operation can be performed by the two disks individually. Write performance and capacity decrease by 50 % due to the redundant data.

Level 10 is a combination of 0 and 1. Two disks run in striping mode and two disks act as mirrors for striped disks. Even though two mirror disks are used, this level only provides single disk failure. Due to the mirroring strategy only 50 % of the disk's capacity and write performance can be achieved.

To overcome this problem *level 4* is introduced. Here k disks are combined and form a group. K-1 disks store the incoming stripe units and one disk stores the parity calculated over all stripe units. Figure 2 shows the logical view of a file, the RAID-0 representation and the RAID-4 representation, including the generation of the parity blocks.

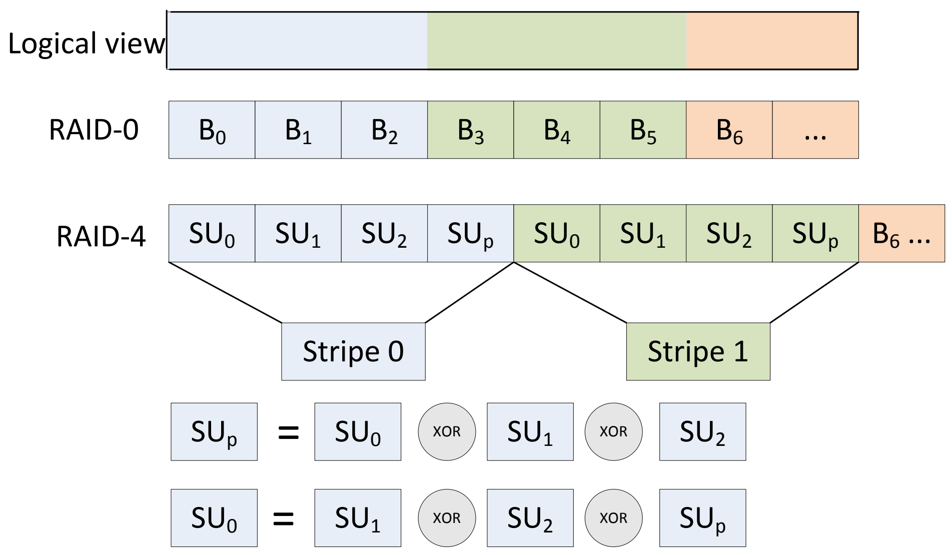

Fig. 2: A files logical view and RAID striping comparison

This level's advantage is the higher efficiency compared to the mirroring approach, while still providing single disk failure protection. The problem with RAID-4 is the parity disk. Every write operation on a stripe unit of the disks 0 to k-2 results in a write operation on the parity disk. For many single block operations the parity disk is becoming a bottleneck. This problem is called *small-write-problem*. Updating the parity block requires reading the old parity block and reading the data block that is going to be updated. Consider a parity unit with current version i: SU_p^i. Then an update of SU_0 with current version j requires the calculation of:

$$SU_p^{i+1} = SU_p^i \otimes SU_0^j \otimes SU_0^{j+1}.$$

Therefore, a small write operation that would require a single disk I/O operation on a striping RAID, here requires four disk I/O operations. At *level 5* the parity block is distributed in a round robin way over all disks, spreading the load on the parity blocks over multiple disks. Nevertheless, the high overhead of the parity block update still exists.

There are several other levels providing double or even triple disk failure protection. Since my system's current implementation is using level 4, I am not going into further detail here.

3.2.2. RAID In Distributed Storage Systems

As mentioned in section 3.1, fault-tolerance is also crucial in distributed file systems. The big difference in striping data at a server level is the lack of a central controller. At level 4 and 5 access to the parity block has to be coordinated. For local disk RAID systems this is done by the RAID controller. At the server level this has to be coordinated by another entity.

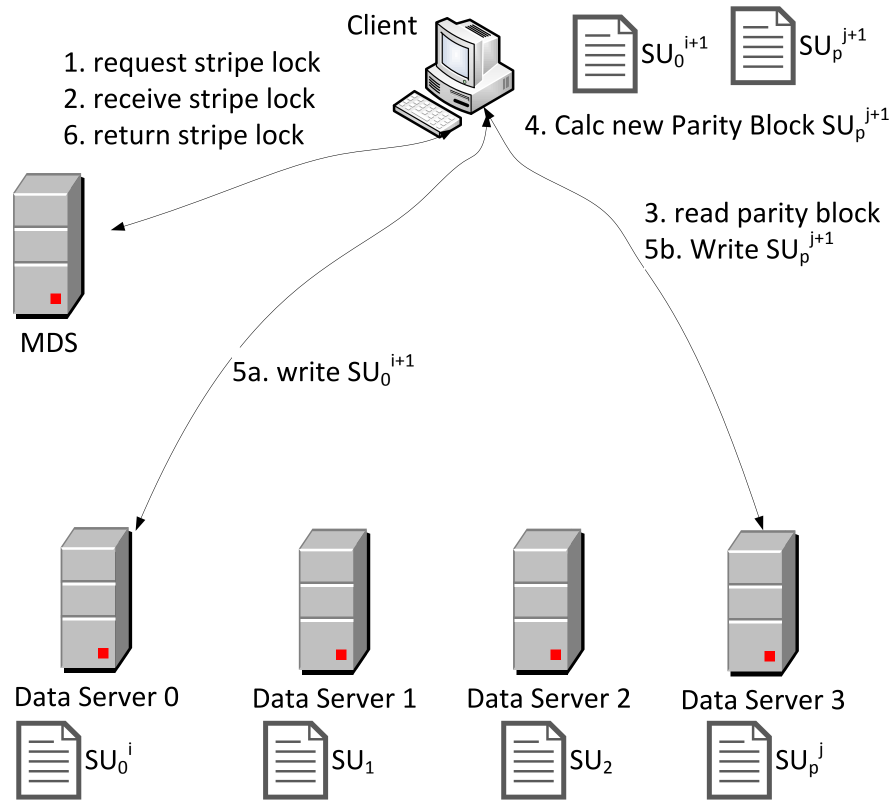

Fig. 3: Client stripe locking during a write operation

In current distributed file systems, the parity calculation is done by the client. In an interactive system two clients might have access to different stripe units of the same stripe and share the stripe's parity block. In order to preserve stripe consistency, access to this parity block needs to be organized. The pNFS standard does not specify the details of this coordination. However, it recommends handling this issue at the metadata server. PanFS[33] uses a technique that basically implements metadata server managed stripe-locks. Figure 3 shows a write operation using a stripe lock.

The system consists of four data servers in a RAID-4 environment. Data server 3 is responsible for the parity block. The client wants to update stripe unit 0 with current version i (SU_0^i) stored at data server 0. At first the client requests a stripe lock from the MDS and receives the lock at step two. Next, the client needs to get the latest version of the parity block SU_p and reads it from data server 3. In this example I consider the client is having a byte-range lock on SU_0 and therefore, has the latest version SU_0^i cached. Otherwise it would have to update SU_0 too. SU_0^{i+1} marks the new data block and ΔSU_0 marks the changes made to SU_0. The client calculates

$$\Delta SU_0 = SU_0^i \otimes SU_0^{i+1}$$

and

$$SU_p^{j+1} = \Delta SU_0 \otimes SU_p^j.$$

At step 5 the new blocks are sent to the appropriate data server 0 and 3. After that the operation is complete and the lock is returned to the MDS.

This is a good example of the problems coming up with distributed file systems. All the coordination between the MDS, the data servers and clients are built-in with a centralized RAID controller. During this read-modify-write cycle the stripe is locked for all other clients, which limits the degree of parallelism. Furthermore, the higher complexity of the distributed workflow both increases the risk of errors and the overhead of error recovery.

4. Coordinated Data Server Cluster

This chapter introduces the coordinated Data Server (DS) cluster. First, the general system environment and the architecture of the metadata server, data server and client are specified. The following sections describe the protocols used to specify the communication among these entities and the resulting workflow. Error handling, the data server replacement mechanism and load balancing considerations are presented and finally, some implementation details are provided.

The focus of this thesis lies on the design of the coordinated cluster workflow and its consistency model. My workflow will be compared with the externally coordinated workflow presented in section 3.2.2. As described in the Related Work section, the pNFS standard is a good example for these. My system will have several features in common with the pNFS standard, but is not designed to be pNFS compatible. Nevertheless, the resulting system could be used to design a new pNFS storage protocol and integrate the data server cluster into a pNFS environment.

4.1. System Environment

My architecture can be considered an extension to the pNFS architecture described in section 3.1.1. It consists of a metadata server, many client systems and a data server cluster. In contrast to the pNFS specification, my data server form a peer-to-peer network and therefore, can coordinate their actions.

The details of the client library, the data servers and the metadata server will be described in the following sections.

4.1.1. Metadata Server

The metadata server has been developed by the project group "Development of a flexible, high-performance distributed file system" at the Paderborn Center for Parallel Computing (PC^2). In this project group we used the ganesha-nfs[3] server as a NFS front-end and designed the metadata server cluster as a back-end. The communication of these components was implemented using the Zero Message Queue ($\oslash MQ$) framework. Since the client session handling was done by the ganesha-nfs server, the MDS development focused on the file system operations. For this thesis I needed my own client library. Therefore, it was more convenient to directly access the MDS' $\oslash MQ$ interface and extend the metadata server's capabilities accordingly.

The metadata server (MDS) architecture itself is quite complex, therefore, I will only explain the components relevant for this thesis, as seen in figure 5.

The *Client Session Manager* returns system wide unique 4-byte ClientSessionID (CSID) to the client. This session ID is used both by the MDS and the data servers to identify the client. During the mount process a client connects

[3]http://sourceforge.net/apps/trac/nfs-ganesha/

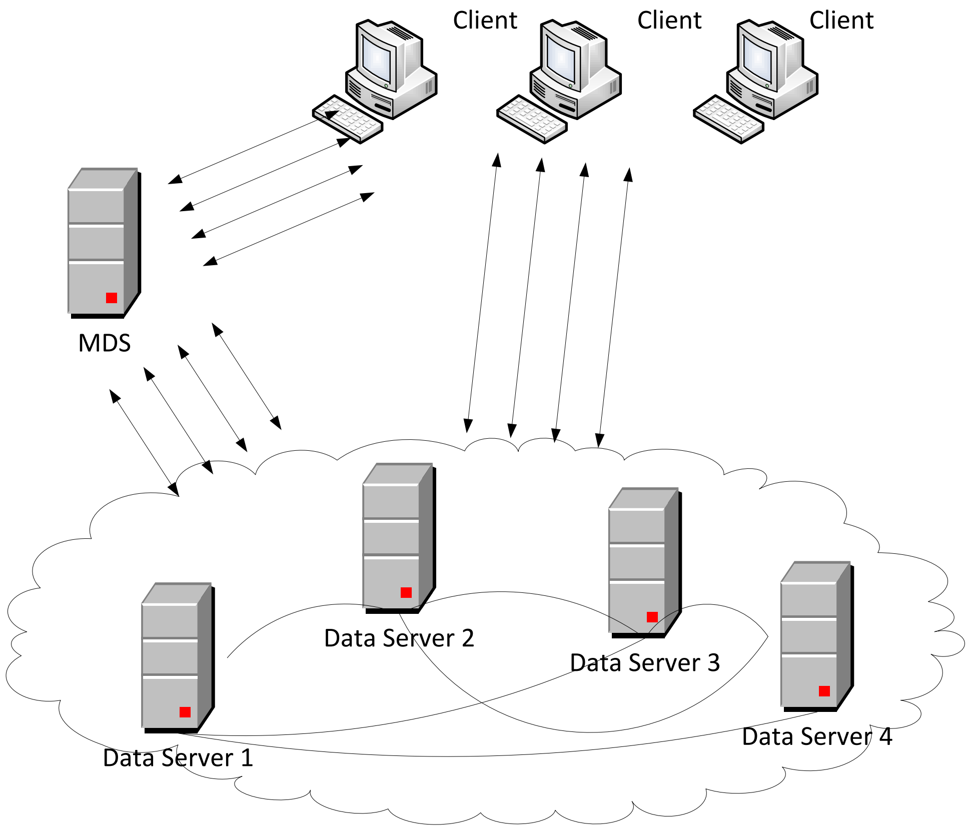

Fig. 4: Coordinated cluster architecture

to the metadata server and request a new CSID. The *Data Server Layout Manager* keeps track of the currently available data servers and their IP addresses. Both the client and the data server use this mechanism to look up the available data servers.

The *Byte-range Lock Manager* organizes POSIX byte-range locks. For the client holding a byte-range lock (BRL), it guarantees exclusive write access to the specified byte-range. These are not to be confused with stripe-locks as described in section 3.2.2. During the evaluation of the stripe-lock mode, I used this component to simulate a stripe-lock mechanism.

When a new file is created, the MDS initializes the file layout. This is done by the *Raidlib* and is based on the specified RAID algorithm. For further details of the file layout management refer to section 4.3.1.

The above mentioned components have been designed and implemented during this thesis. The following components were designed and implemented by the project group. All the components that are involved in the actual file system management are represented by the *Journal / Cache* component. These are responsible of handling the file system operations, such as reading a directory, creating or deleting a file system object. The *Storage Abstraction Layer*

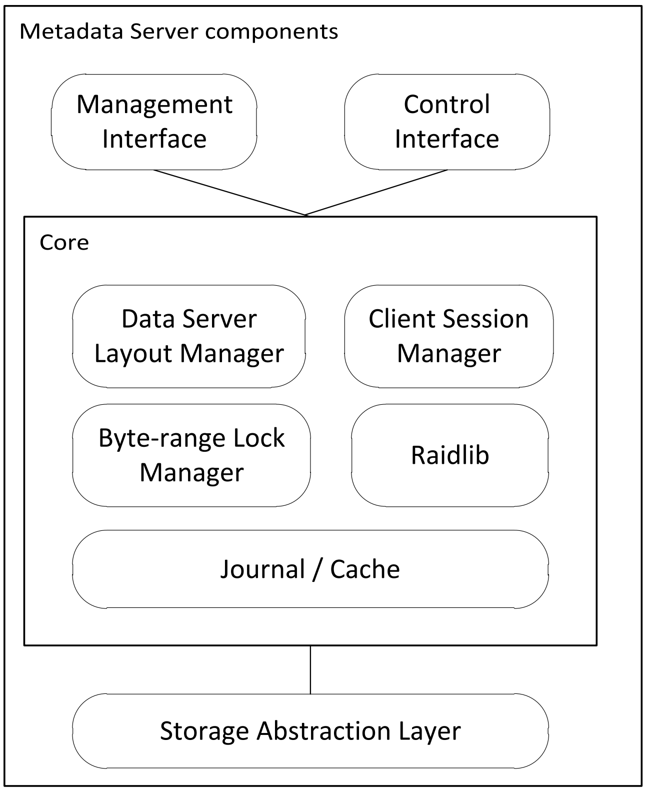

Fig. 5: Metadata Server Components

represents the interface to the permanent storage.

The MDS runs two network interfaces that both serve a synchronous request-response broker pattern shown in figure 6. It is implemented using the $\oslash MQ$[4] framework. The broker queues the incoming client requests and the MDS worker threads process them in a First-Come-First-Serve manner. The worker threads return the response to the broker and this forwards the message to the waiting client. Currently, there is no backchannel necessary for the MDS to send requests to the clients or data servers. Besides the improved degree of parallelism, one design goal of my cluster architecture is the reduced load on the metadata server. Ideally the MDS stays passive, offers its services in a remote procedure call (RPC) style and does not need a backchannel. All tasks that require active participation of the MDS, such as timeout monitoring or load balancing, should be handled by the data server cluster.

Client requests are taken care of by the *Management Interface* and the *Control Interface* is accepting data server requests. Details on the served protocols can be found in section 4.2.

[4]http://www.zeromq.org

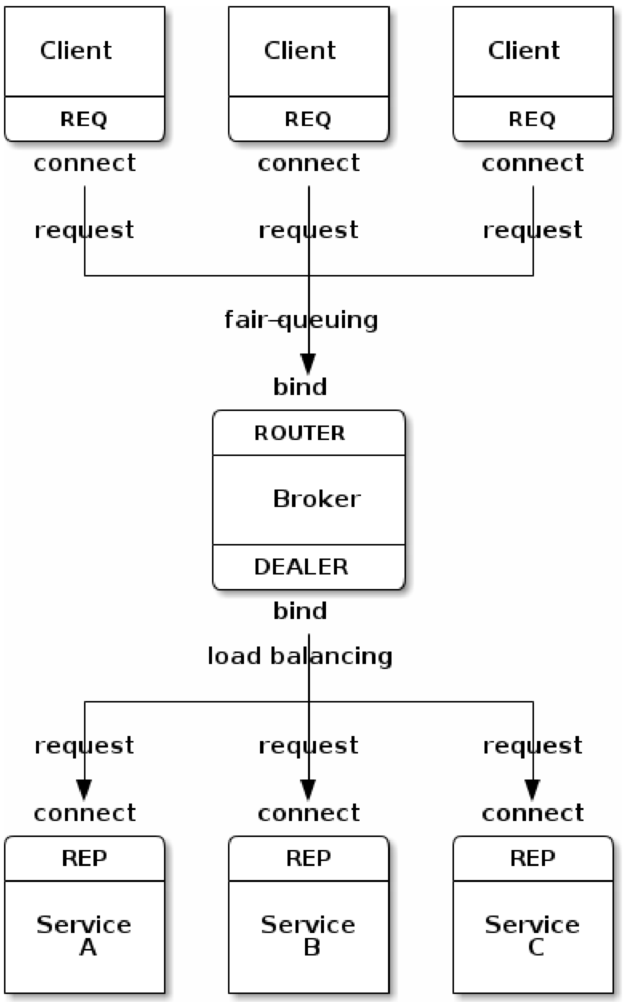

Fig. 6: Request-response broker pattern used by the MDS[a]

[a] https://github.com/imatix/zguide/raw/master/images/fig20.png

4.1.2. Client Library

The client library should be integrated into the system's user space via the FUSE API. Currently the FUSE support is not implemented yet. For further details on this issue refer to sections 4.8 and 5.2.1. Figure 7 shows the client library's components.

Two communication protocols are handled. The *Management Interface* interacts with the metadata server. This covers management task such as session management, lock management, file system operations and data server layout look ups. The *Storage Interface* interacts with the data server and is primarily used to perform data read and write operations. The Storage Interface connections work unidirectional. Therefore, the client opens an asynchronous server instance to provide a backchannel for the data server cluster.

Incoming user requests are forwarded to the *Operation Manager*. This ana-

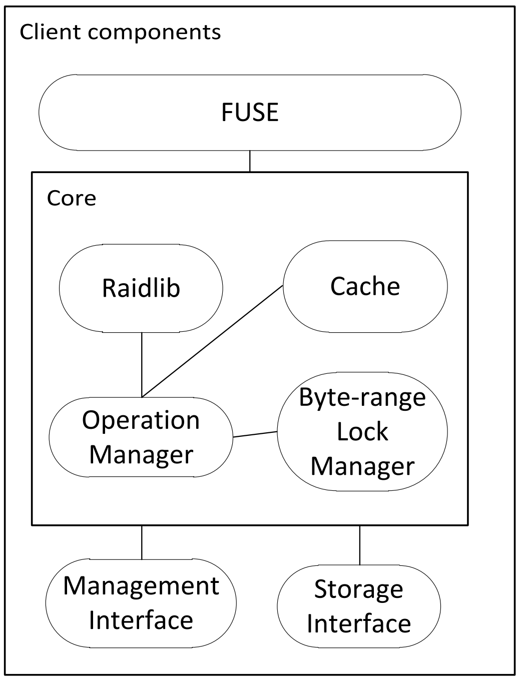

Fig. 7: Client library components

lyzes the request and identifies the necessary steps to perform the operation. This includes the identification of the involved stripes and splitting the client request into a set of sub-requests, which cover the identified stripes individually. The Operation Manager monitors each client request, identifies timed out sub-requests and reissues timed out sub-requests if necessary. Once a defined time passed, an error message is returned.

The *Raidlib* is used to read the file layout and identify the participating data servers. If some of these data servers are unknown to the client, it requests the needed information from the MDS using the Management Interface. The above mentioned creation of sub-requests is based on information provided by the Raidlib.

The *Cache* is used to store data objects that are covered by an BRL. In section 3.2.2 I described the read-modify-write issue of parity based RAID systems. Only the parity block had to be read, whereas the old version of the modified block was considered to be cached. Without a client cache the current version of the modified data block would have to be read at step three, too. For shared data objects a cache invalidation protocol needs to be implemented. This is not covered during this thesis. The *Byte-range Lock Manager* organizes the allocated locks and monitors the lock's lease times.

4.1.3. Data Server

Figure 8 displays the main components of the data server. The *Control Interface* interact with the metadata server. This is primarily used to perform data server layout look ups. Further coordination between the data server cluster and the metadata server cluster is needed for load balancing and recovery operations. The *Storage Interface* represents the server-side of the client's Storage Interface and therefore, handles client read and write requests. The data server internally communicate via the *Cluster Interface*. These are unidirectional channels and requires a contacted data server to create a backchannel in order to respond. The channels are designed to be keep alive during idle times.

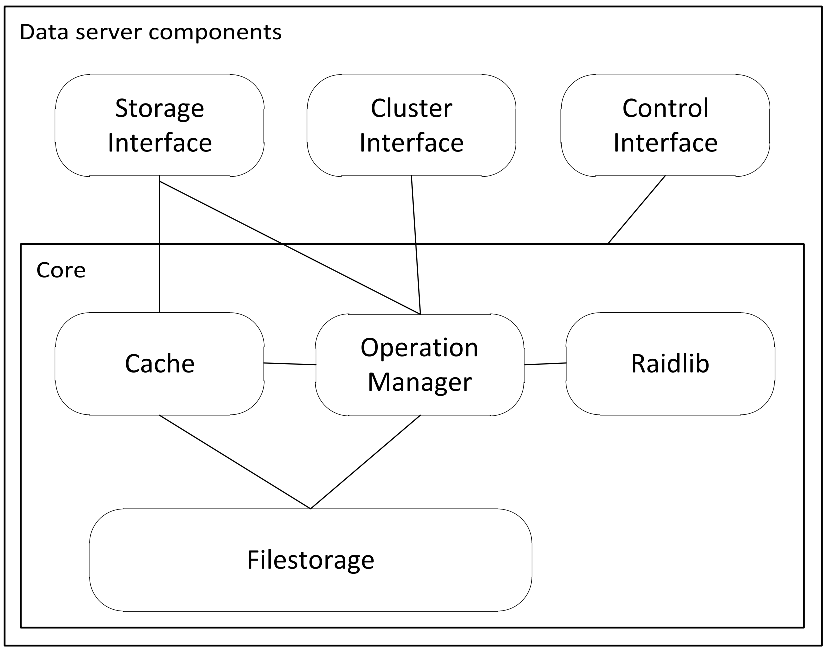

Fig. 8: Data server components

The coordination of the single client operations is done by the *Operation Manager*. This coordinates the execution of each operation, identifies conflicts and performs timeout monitoring. The details of these tasks are described in the sections 4.3, 4.6 and 4.4.

The Operation Manager uses the *Raidlib* to identify the participants and the coordinator of an operation. The *Cache* stores data blocks read from disk and the various unconfirmed data during an operation. Details on the tasks performed by the Operation Manager and the Cache will be explained inside the consistency model 4.3 and workflow 4.3.3 sections.

Data is written to disk using the *Filestorage* component. Similar to PVFS2 or Ceph, it stores data as files using the hosts file system. Currently, I am using a simple three-level tree structure. Starting from a defined base directory, the first sub-directory defines the InodeNumber, the second level defines the stripe ID. In that directory files named after the version of the stripe unit are stored. The data of SU^i, of stripe j, of file k would be written to file *.../k/j/i*. I chose this strategy in order to abstract from the low level data handling and concentrate research on the consistency model. Ext3 provides good performance, but has a maximum number of 32.000 sub-directories. Therefore, in future development a more modern file system, such as btrfs (used by Ceph), should be evaluated.

4.2. Protocols

The system has four different network protocols, which are described in this section. However, this can not become a full API reference, but will provide a basic understanding of the functionality.

4.2.1. Management Interface Protocol

The Management Interface Protocol specifies the client $\leftrightarrow$ MDS communication. The file system operations marked with a '*' were already defined during the project group. The communication is implemented in form of a synchronous request-response pattern and offers the following functionality:

- **CreateSession()**
  The client requests a session id from the metadata server. The return value is a globally unique CSID.
- **AcquireByterangeLock(CSID, Offset, Length, InodeNumber)**
  The client requests exclusive write access to a part of a file specified by its InodeNumber. Offset specifies the start of the lock area and length the size. A failure is returned if the requested range conflicts with an existing BRL otherwise success is returned.
- **ReleaseByterangeLock(InodeNumber, Offset)**
  The client returns a previously acquired BRL. The metadata server identifies the lock by the InodeNumber and offset.
- **LookupName(InodeNumber, Name)***
  This is a file system operation. The client looks up a name within the directory identified by the InodeNumber. Returns true if a file system object matching the name exists and false otherwise.
- **Readdir(InodeNumber)***
  The client reads all file system object's metadata that are found inside the provided directory. Depending on the directory's size many response messages might be necessary.
- **ReadEinode(InodeNumber)***
  The client reads the metadata of an existing file system object. In our project group's MDS, the metadata of an object is called an Einode.

- **CreateObject(InodeNumber, Name, Gid, Uid)***
  The client creates a new file system object inside the directory specified by the InodeNumber. The newly created object's InodeNumber is returned.
- **UpdateObject(InodeNumber, set of attributes)***
  The client updates attributes of a file system object. Among these are the user and group ID, file mode, file layout and size. The return value indicates success or failure of the operation.
- **RemoveObject(InodeNumber)***
  The client deletes a file system object. The return value indicates success or failure of the operation.
- **DataserverLayout()**
  The client requests a list of all data server IDs registered at the metadata server.
- **DataserverAddress(ID)**
  The client requests the address of a specific data server defined by its ID.

4.2.2. Control Interface Protocol

The control interface protocol defines the communication between the data servers and the metadata server.

- **DataserverLayout() and DataserverAddress(ID)**
  The data server need to lookup the addresses of each other. Therefore, the two lookup operations known from the Management Interface Protocol are used as well.
- **UpdateObject(InodeNumber, set of attributes)**
  During a recovery process or for load balancing reasons it might be necessary to migrate data from one data server to another. This requires an update of the object's file layout.
- **DataserverLoadStatus(attributes)**
  When a file is created, the metadata server determines the striping of the data among the data server. Therefore, information on the current load of the data server cluster should be part of the placement decision. The load balancing mechanism and the corresponding attributes have not been defined yet.
- **DataserverStatus(ID)**
  The data server cluster identifies unavailable nodes. If the cluster declares a data server to be unavailable, the MDS must be informed too. The MDS is not directly part of the recovery process, but should not assign newly created files to that data server.

4.2.3. Storage Interface Protocol

The storage interface protocol specifies the communication between the client and the data server. The following operations are implemented:

- **Read(OpID, InodeNumber, Offset, Length)**
  The client reads the specified range of a file. Over the backchannel the requested data is sent by the data server. The OperationID (OpID) is a combination of the clients CSID and a sequence number generated by the client's Storage Interface. It is globally unique.
- **WriteStripe(OpID, InodeNumber, Offset, Length, Data)**
  The client writes a complete stripe. This message indicates that the client also writes the parity data of that file. The data server guarantee the atomicity of the write operation, but do not need to handle parity calculation. The success of the operation is returned over the backchannel.
- **WriteSU(OpID, InodeNumber, Offset, Length, Data)**
  The client writes a single stripe unit. The data server needs to calculate the operation delta and coordinate the update to the parity block with the parity data server. The success of the operation is returned over the backchannel.
- **DirectWrite(OpID, InodeNumber, Offset, Length, Data)**
  This request directly writes the data to disk. No coordination takes place. This mode is only used during the evaluation of the stripe-lock mode. The success of the operation is returned over the backchannel.
- **StatusRequest(OpID)**
  The client suspects a timeout of an operation. The status report is a request to identify the state of the operation on a particular data server.

4.2.4. Cluster Interface Protocol

The cluster interface protocol specifies the interaction of the data server cluster. These use unidirectional communication, too. Some of the following messages might be hard to understand without knowledge of the workflow described in section 4.3.3. The cluster communication is necessary for the coordination of client write operations. The server, which is responsible for the parity block of a stripe acts as the coordinator of the operation. Furthermore, the cluster uses a global version management mechanism to reason on the stripe's state.

- **Received(OpID, InodeNumber, Offset, Length, File layout)**
  The data server received a write request from a client. This message is used to inform the coordinator about the new operation and indicates that the sender is ready to perform it.
- **Perform(OpID)**
  The coordinator decided that the operation, identified by OpID, is ready to be executed. It informs all participants which previously sent the corresponding *received* message.

- **Cancommit(OpID, ΔSU, Versionnumber)**
  ΔSU represents the difference of the new data received from the client and the existing data. It is needed by the parity server to restore the stripe's consistency. The version number is used to update the stripe's version vector.

- **Docommit(OpID, Versionvector)**
  The coordinator informs all participants to write the new data to disk. The version vector represents the stripe's state and was updated by the *cancommit* messages sent previously to this one.

- **Committed(OpID)**
  The data server wrote the data to permanent storage. This message guarantees to the coordinator that the operation has been performed by the sending data server.

- **Success(OpID, Status)**
  Sent from the coordinator to all participants. Either the coordinator received all committed message, then the operation was successful, or something went wrong and the operation was not successful.

- **Dismiss(OpID)**
  An operation can not be performed and needs to be dismissed. If no data has been written to disk, the data server simply removes the operation data. No further coordination is needed.

- **Dismissed(OpID)**
  If however, data had been written to disk, the coordinator has to guarantee the stripe consistency. Therefore, the data server must reply to an incoming dismiss message with an dismissed message in order to confirm the clean up operation.

The load balancing mechanism was not part of this thesis and therefore, has not been specified yet. Some ideas of the cluster's load balancing potential are presented in section 4.7. The detailed recovery messaging has not been specified and implemented too. I found little scientific value in implementing the recovery process. The new data server replacing a failed one is basically acting as a client and reading the stripe units from the other servers. Then the lost data is recovered. I discuss the workflow and recovery process issues in section 4.3.5.

4.3. Consistency Model

This section explains the consistency model. Scalability, availability and performance are desired qualities in distributed storage systems. A popular measure to specify the system's availability rate is the x-nines term[17]. The x represents the number of leading nines of the availability rate. For example a 5-nines system provides an availability rate of 0.99999. This would result in an annual downtime of about 5,2 minutes. However, consistency is a quality, which cannot be relaxed in file system design. Returning inconsistent data is worse than returning none. Whereas a 100.0 % availability cannot be guaranteed, consistency must be ensured by my architecture.

4.3.1. Definitions

Before we can have a look at the consistency model's details, we need to clearly define the involved objects.

0 31 63

type	gsize	server count	susize
id_0			id_1
$id_{2,..,61}$			

Fig. 9: RAID 4 file layout

The *file layout* displayed in figure 9 specifies the distribution of the data blocks to the data server. The first byte defines the type of the layout and the second defines the size of the individual server groups. In this thesis the RAID-4 algorithm is used. Therefore, a groupsize of 9 would define a 8+1 striping. The next two bytes represent the number of servers defined in this file layout. Currently the file layout has a total size of exactly 256 bytes. At most 62 4-byte server ids can be specified. The server count defines the number of valid ids. It must be a multiple of the groupsize parameter. *Susize* specifies the size of a single stripe unit. The following bytes are used to define the single data server ids that the file is striped across.

0 63 127

1	9	18	1048576	0	1
2			3	4	5
6			7	8	9
10			11	12	13
14			15	16	17
...					

Fig. 10: Example of an RAID-4 file layout

Figure 10 shows an exemplary file layout. It specifies 18 data servers and groups them into two groups. Let's assume that type=1 specifies a RAID 4 file layout. Then the servers 0..7 and 9..16 store the actual user data and server 8 stores the parity of 0..7. Server 17 handles the parity of 9..16 respectively. The stripe unit size has been set to 1048576 bytes = 1 megabyte (MB). Currently, the striping is implemented in a round robin fashion. That means the first 8 MB form the stripe 0 and are written to the first group 0..7. The next 8 MB of a file are stored by the second group and the stripe from MB 16 to MB 24 are stored by the first group again.

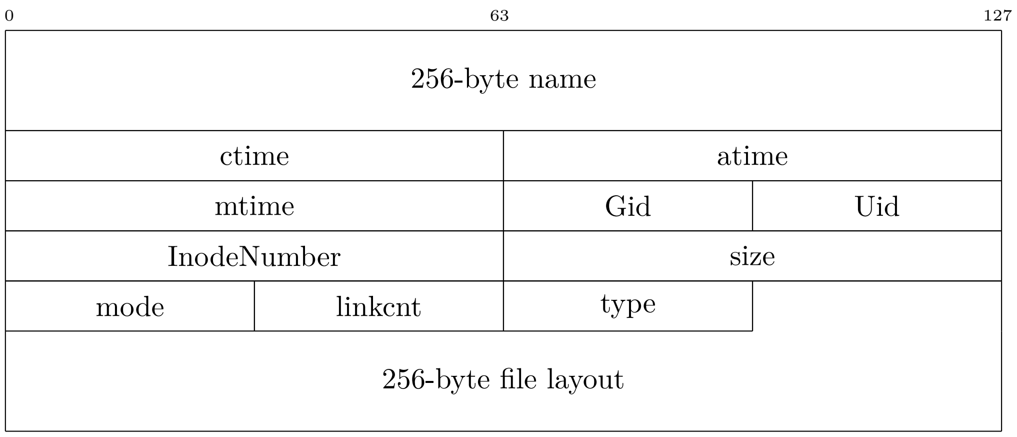

Fig. 11: Einode attributes

The Einode contains the file's metadata. It is managed by the metadata server and requested by the client. All the attributes, except for the name, form the Inode. The Inode is part of every client read and write request. Each data server needs the *file layout* attribute in order to identify its role within the operation and look up other participants, such as the parity server. The InodeNumber is used to identify the file system object.

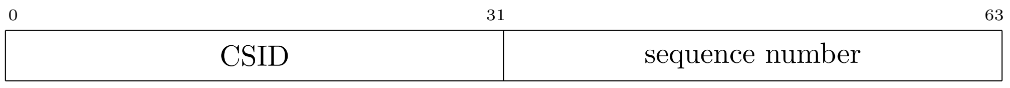

Fig. 12: Operation ID (OpID) specification

Figure 12 shows the structure of an OpID. The Client acquires its CSID during the mounting process. This 32 bit unique id is assigned by the MDS. The sequence number is managed by the client library and incremented at every operation. The CSID is unique throughout the system and the sequence number is unique for every operation of that client. Therefore, the uniqueness of the OpID is guaranteed.

Figure 13 shows the structure of a data object (DO) stored by a data server. The *OpID* is used to identify the Operation that lead to the creation of that data object. The attribute's *offset* and *length* specify the range of that operation within the file. These attributes are required for an integrity check of the stripe. The version entries *Version* DO_i are 32 bit unsigned integers. These version

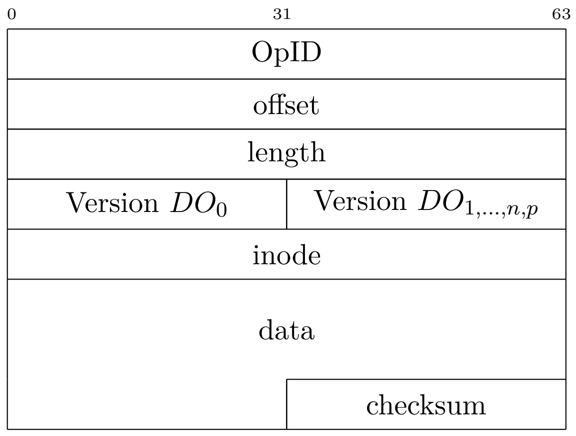

Fig. 13: Data object structure definition

numbers specify the global state of the stripe. Further details on the global state management will be described in section 4.3.2. *Data* contains the actual file data of this object.

The inode is part of the data object, too. After, a cluster failure occurs, each data server must perform an integrity check on its stored data. In order to accomplish this, it needs the file layout, which is part of the inode. This could be acquired individually from the metadata server, but this would produce a high load on these servers during a system-wide integrity check and would require a version management mechanism for the file layout. Considering a stripe unit size of 1 MB, I found the overhead of a few bytes per data object acceptable.

The checksum represents the XOR parity of all metadata attributes. A data object is considered consistent if the checksum is valid. In a RAID 4 system the stripe object (SO) consists of several data objects with one storing the parity over the others and therefore, providing single failure tolerance. Each of these is stored on a different data server. Without loss of generality let n denote the number of data blocks per stripe and therefore, provide a $n+1$ redundancy model. DO_i shall be a data object stored on a data server. A stripe object SO can be defined as follows:

$$SO = \{< DO_0, ..., DO_n >| \; DO_n = DO_0 \otimes \ldots \otimes DO_{n-1}\}$$

A stripe object is considered consistent, if all data objects are consistent and the parity block DO_n represents the parity of all data objects. A file object (FO) is an ordered set of stripe objects:

$$F = \{< SO_0, ..., SO_n >| \; \forall i : SO_i \text{ is consistent } \}$$

4.3.2. Global State

In a distributed file system, managing the file's state in a centralized manner does not scale and creates a single point of failure. A stripe object's state is determined by the state of the individual data objects. Each update causes the data object's version to be incremented.

At this point it is sufficient to know that during an operation the parity server acts as the coordinator. Therefore, I will concentrate on the version changes caused by the client operations. The following example describes the global state management during a client write. For simplicity we focus on a single stripe object in a 3+1 RAID-4 configuration. Figure 14 shows the state of the data objects DO_0 as seen by DS_0 and DO_1 as seen by DS_1.

0 31

Version $DO_0 = 22$
Version $DO_1 = 25$
Version $DO_2 = 11$
Version $DO_P = 34$

(a) State of DO_0 on DS_0

0 31

Version $DO_0 = 22$
Version $DO_1 = 26$
Version $DO_2 = 11$
Version $DO_P = 35$

(b) State of DO_1 on DS_1

Fig. 14: Data object states as seen by data server 0 and 1 within the same stripe object

The current version of DO_0 is 22 and of DO_1 is 26. All we know about the state of DO_2 and DO_P is that their version indices are at least 11 and 35, respectively. DO_0 denotes the state of DO_1 to be 25, whereas the actual version is 26. Therefore, there must have been a write operation on DO_1 causing the version increment to 26 and the increment of the parity block to 35. DS_0 was not part of that operation and is unaware of it happening.

A initial create operation followed by a partial update can be seen in figure 15. It shows a stripe object with n=4 and DS_3 storing the parity data object. The block to the right displays a shortend version of DO_0's metadata created by operation 43.

The client operation (OpID=42) creates the file and operation OpID=43 updates DO_0 and DO_1. Here the stripe unit size is set to 100.000 bytes and therefore, an operation with offset=0 and length=200.000 covers DO_0 and DO_1. The first cut represents a point in time, which is after operation 42 finished successfully and before operation 43 started. A client read operation on the whole stripe would return a valid state, since all data server return the initial stripe's state. The third cut also represents a valid state, since operation 43 has been performed by all data servers.

The second cut is inconsistent. Here a client read would return DO_1 version 1, although DO_0 and DO_p already return their new versions. Basically, there are two solutions to this problem. The first would solve it at the data server cluster by coordinating the version switch at the cost of additional latency and more failure cases. The other one handles this issue at the client side. There it is impossible to prevent the race condition, but it can be detected. The client reads the metadata of DO_0. Offset and length indicate that DO_1 has been

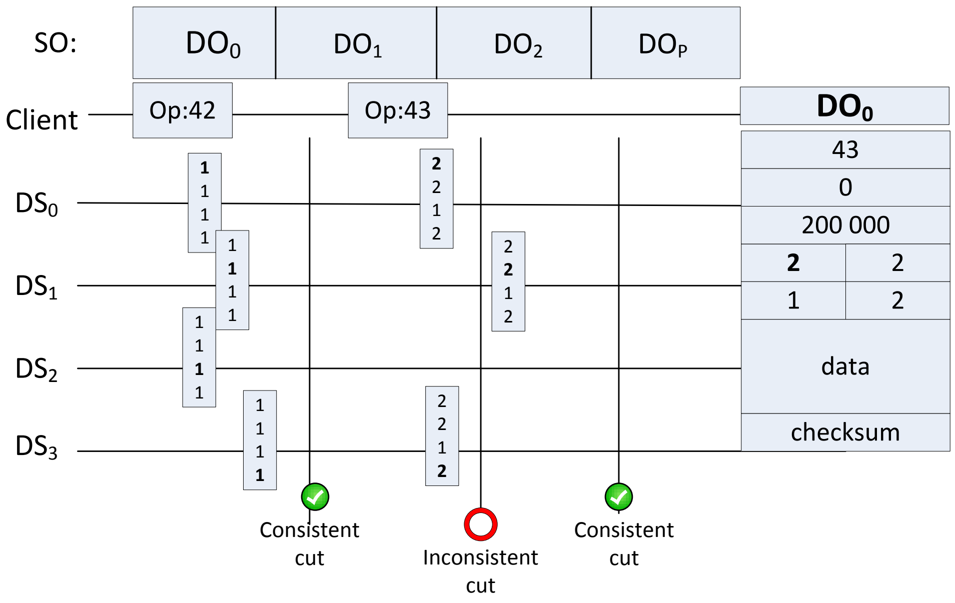

Fig. 15: Consistent and inconsistent cuts

part of operation 43 and looking at the version vector, the client knows that DO_1's version should be at least 2. There could have been an operation after 43 modifying DO_1 that DS_0 is unaware of. Next, the client reads the metadata of DO_1 and recognizes that DO_1 is outdated. It requests the new version from DS_1 and restarts the validation process. DS_2 was not part of operation 43 and its version index was not incremented.

4.3.3. Write Operation Workflow

Now it is a good time to investigate the workflow in more detail. During a write operation the data server responsible for the stripe's parity block acts as the coordinator. Figure 16 visualizes the coordination process between a client and three data server. The client performs an operation that covers the data server DS_0 and DS_1.

First, the client sends the new data objects DO_0 and DO_1 to the appropriate data servers. The required metadata of that operation (OpID, offset,length,etc.) are represented by *op*. DS_0 processes the request, identifies the coordinator DS_p and send the *received(op)* message. This message tells the coordinator that DS_0 received the operation op and is able to process it. Due to the *offset* and *length* attributes, the coordinator know that DS_1 is also part of this operation and postpones further processing. Once all *received* messages are collected, the coordinator starts phase 1 and broadcasts the *perform(op)* message to all participants. Each data server now reads its old data block and calculate

$$\Delta DO_i = \text{new } DO_i \otimes \text{existing } DO_i$$

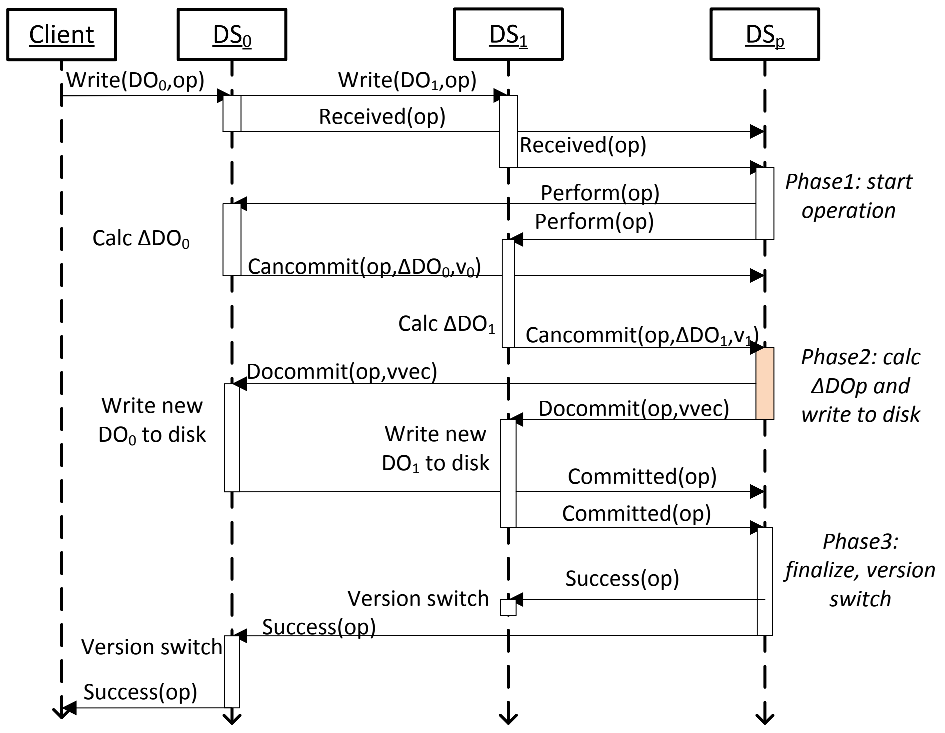

Fig. 16: Coordination of a write operation to two stripe units

Along with the new version number for this data object ΔDO_i is returned to the coordinator. This collects all *cancommit* messages and enters phase two. Here the new version vector is then multicasted using the *docommit* message. The coordinator itself calculates the parity object

$$DO_p^{j+1} = DO_p^j \otimes \Delta DO_0 \otimes \Delta DO_1$$

and writes the new object to disk. This second phase on the coordinator is the part of the workflow that needs to be serialized. Once the data block DO_p^{j+1} has been created and written to disk, a parallel operation can create DO_p^{j+2}.

The participants create their new version based on the received version vector in the *docommit* message and the data initially received from the client. They write this to disk (including *fsync()*) and send the *committed* message to the coordinator. The flush at the participator side guarantees that the data is written to non-volatile storage before the committed message is sent.

Once all committed messages arrived, the coordinator synchronizes its file handle and multicasts the *success* message. This message guarantees that the operation has been performed by all data servers and therefore, the operation is considered to be performed successfully. Now the coordinator will answer read requests with the new version DO_p^{j+1}. The participants receive the success message and will also switch to the new version. DS_0 is the data server with the lowest index within this operation and therefore, forwards the success message to the client. The operation has been completed.

4.3.4. Version Management On Disk

Coordinating distributed write operations also means having to perform undo operations if one participating data server fails. Especially, the parity object will have to deal with multiple parallel operation and therefore, handle several versions of the same data object with unconfirmed status. Unconfirmed status means that the data object has been written to disk, but not all *committed* messages have been collected from the participants.

The data blocks are stored inside the hosts file system. The top level directory specifies the InodeNumber and the sub-directory represent the stripe's id. Inside the stripe id's directory, the different data object's versions are stored as files and are named by their version number. Therefore, version 1 of stripe id 0, of a file system object with InodeNumber 42, would be stored as .../42/0/1. A new version would then be stored as file name 2. My architecture uses the copy-on-write (COW) mechanism. This means that multiple versions of the same stripe id's data object are allowed to exist. During the version switch from 1 to 2, the now outdated file 1 will be added to the garbage collector and deleted at a later time.

4.3.5. Recovery After System Crash

While the garbage collector works great if all necessary information are cached, a system crash complicates matters. There is no log or centralized journal that could describe the state of each data object version found in a directory. Figure 17 shows a system crash during the third operation 66.

After its reboot, DS_0 sees two data objects for the represented stripe. First, it checks all data server that were part of the operation that created version 1. All of them already removed their version 1 objects, therefore, the operation has been performed successfully and DS_0 marks version 1 to be valid. Then it checks version 2. Both DS_1 and DS_3 have their version 2 blocks stored on disk. This indicates that operation 43 has been performed successfully and DS_0 marks version 2 to be valid and version 1 can be garbage collected. Let's assume DS_2 and DS_3 already agreed on operation 55 and DS_1 send a request to DS_3 to check its version 3. Operation 66 has been interrupted by the system crash. In this example only one part of the operation is missing and can be recovered using the parity data. DO_2^3 is missing, but can be recovered:

$$DO_2^3 = DO_0^2 \otimes \mathrm{DO}_1^3 \otimes \mathrm{DO}_p^4$$

If multiple data objects of an operation are missing, the other ones need to be removed too. Should DO_1^3 be missing too, and only DO_p^4 of operation 66 were recovered, removing DO_p^4 would create a consistent state. Removing this operation is valid, since the operation has not been confirmed to the client yet.

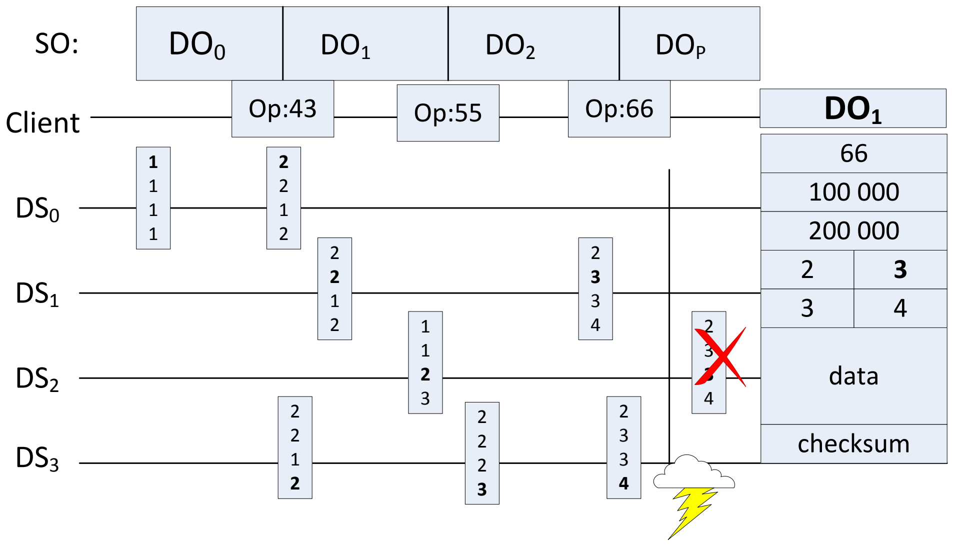

Fig. 17: Recovery after a system crash

4.3.6. Removing Interleaved Operations

Identifying the state of the stripe in the above example was straight forward. Figure 18 shows three operations with only the second one failing.

The situation is as follows. Operation 43 has been performed successfully. Operation 55 is currently in the second phase. The parity object DO_p^3 has been written to disk, but the participants did not respond. From the coordinator's point of view, the operation is unconfirmed and it is waiting for the results from DS_0 and DS_1. It continues to perform operation 66, issued by DS_2. After this operation is confirmed, the timeout hits operation 55. At this point DS_p manages three versions (2, 3 and 4) of the object. DO_p^2 and DO_p^4 are confirmed, but DO_p^3 is invalid due to the timeout. Let DO_p^1 be the version before operation 43:

$$DO_p^2 = DO_p^1 \otimes \Delta Op(43)$$
$$DO_p^3 = DO_p^2 \otimes \Delta Op(55)$$
$$DO_p^4 = DO_p^3 \otimes \Delta Op(66)$$

Since operation 55 is invalid, version 4 must represent the operation

$$DO_p^4 = DO_p^2 \otimes \Delta Op(66)$$

whereas it currently represents:

$$DO_p^4 = DO_p^2 \otimes \Delta Op(55) \otimes \Delta Op(66)$$

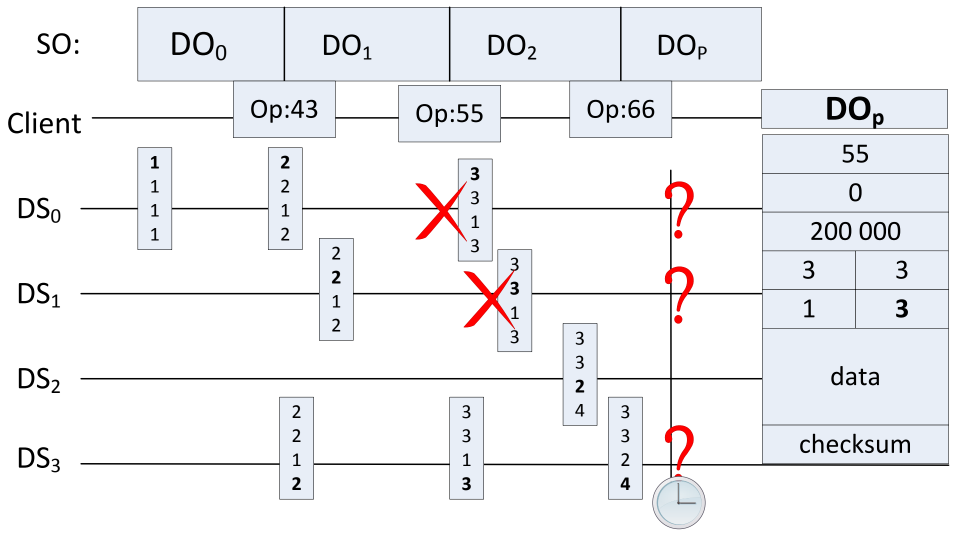

Fig. 18: System state that needs to undo operation 55 while 66 has been confirmed.

To complicate matters, we assume a system crash and all cached data is lost. This means we do not have direct access to the $\Delta Op(xy)$ data, but do have access to DO_p^2, DO_p^3 and DO_p^4. The correct data object is created in the following way:

$$\begin{aligned} DO_p^3 &= DO_p^2 \otimes \Delta Op(55) \\ \Leftrightarrow DO_p^3 \otimes DO_p^2 &= \Delta Op(55) \\ oldDO_p^4 &= DO_p^3 \otimes \Delta Op(66) \\ \Leftrightarrow oldDO_p^4 \otimes DO_p^3 &= \Delta Op(66) \\ \Rightarrow newDO_p^4 &= DO_p^2 \otimes oldDO_p^4 \otimes DO_p^3 \end{aligned}$$

The new DO_p^4 version is written to disk as file name *4r*, since the invalid version's file *4* file must not be overwritten. Consider a system crash immediately after $newDO_p^4$ is written to file 4. Then a subsequent recovery operation would generate $oldDO_p^4$ again. Once file *4r* is stored permanently, file *4* can be garbage collected.

Furthermore, an empty file "3_invalid" is created to mark the version 3 to be invalid. A version request for an index lower than the lowest available one is considered to be garbage collected and in consequence valid. Imagine DS_1 created the data object, but the response arrived late. A subsequent request on the state of operation 55 to DS_p would result in a valid operation if version 2 has been garbage collected and version 3 were not found either. Then only version 4 exists and version 3 is considered to be garbage collected, whereas it was invalid.

4.3.7. Parallel Operation On The Non-Parity Data

Up to now only the coordination of conflicting operation on the parity object has been covered. The model allows the parallel modification of the same data objects, but the degree of parallelism is limited. A data server might issue several *received* messages for different client operations, but the case from the previous section must never occur.

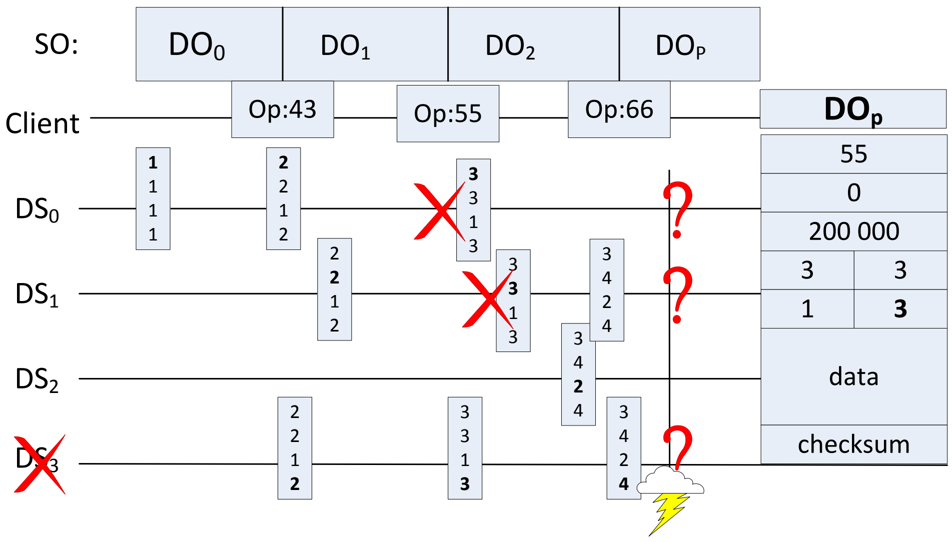

Fig. 19: Non-recoverable system state

Figure 19 shows a stripe that must undo an operation that might have been confirmed to the client. The operation 66 was successful, however, OpID 55 was not. After the crash the parity server is unavailable. Without the parity information the data server cannot scrub operation 55 and therefore, have to return to version 43. The non-parity object need to be written to disk in a strictly sequential manner.

As a compromise a data server, managing two conflicting operations A and B, could base the $\Delta Op(A)/\Delta Op(B)$ operations on the current version i. The data server sends the *cancommit* message for both operations to the coordinator and performs the one of them getting the *docommit* message first. Let's assume operation A is performed. Once A has been finished successfully, operation B would have to be restarted. This time $\Delta Op(B)$ would be based on the data object i+1, as created by operation A.

On the one hand, this workflow creates an overhead due to the parity calculation and the network communication. On the other hand, the first operation ready to perform is scheduled. A strictly FCFS scheduling could block operation B if A is going to time out.

4.3.8. Fsync Delay Issue

The parity object is flushed to disk once phase three is entered. I chose this optimization to give the file system and disk controller a chance to optimize write operations. However, this delayed flush introduces the possibility of data loss.

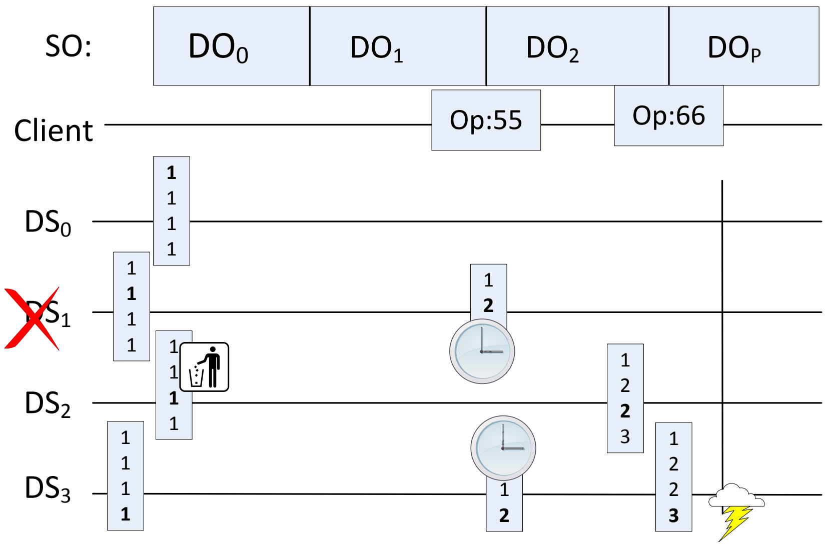

Fig. 20: Missing data objects due to missing fsync call

Figure 20 shows a sequence of events that result in an unrecoverable state. After the initial stripe creation DS_1 performs Op(55) and enters phase two, but does not complete. This means that the parity server was not forced to flush DO_p^2 to disk. DS_2 performs Op(66) successfully. This means, both DO_2^2 and DO_p^3 are flushed to disk. After DO_2^1 is garbage collected, a global system crash occurs.

Neither DO_1^2 nor DO_p^2 have been written to disk. The system crash permanently destroyed DS_1, the other data servers reboot successfully. Since only one data server failed, the system should be able to restore a consistent state and continue to operate.

DO_p^3 currently contains $\Delta Op(55)$:

$$DO_p^3 = DO_p^1 \otimes \Delta Op(55) \otimes \Delta Op(66).$$

Operation 55 was not performed successfully and must be removed from the parity object. Since DO_p^2 is not available, the parity server can not scrub this operation on its own:

$$\text{new } DO_p^3 = DO_p^1 \otimes \text{old } DO_p^3 \otimes DO_p^2.$$

Furthermore, DO_1^2 is not available and therefore, can not be used to reconstruct DO_p^2:

$$DO_p^2 = DO_0^1 \otimes DO_1^2 \otimes DO_2^1.$$

If $\Delta Op(66)$ were available, the correct parity object could be reconstructed based on DO_p^1. Since DO_2^1 already was garbage collected, $\Delta Op(66)$ is not available:

$$\begin{aligned} \Delta Op(66) &= DO_2^1 \otimes DO_2^2 \\ DO_p^3 &= DO_p^1 \otimes \Delta Op(66) \end{aligned}$$

However, DO_2^1 could be reconstructed, but DS_1 failed:

$$DO_2^1 = DO_0^1 \otimes DO_1^1 \otimes DO_p^1.$$

In conclusion the complete stripe is permanently inconsistent. The delayed parity object flush allows this scenario to happen and therefore, must be prevented. A possible solution would be the implementation of a low-level file handle manager. Before flushing DO_p^3, this manager would flush DO_p^2 first. However, in my experience the delayed flush only increased the throughput by about 0-3 % and does not seem to be worth the overhead.

4.4. Handling Errors

During the last section some error cases have already been discussed. Here the workflow is analyzed in more detail. Generally, only single server failures are investigated, since the system only provides single failure tolerance.

4.4.1. Data Server Roles

There are three roles a data server can have. The parity server takes the role of the (primary) coordinator and the others are participants of an operation. Since the coordinator in this case becomes a single point of failure, the participant with the lowest stripe unit index involved in an operation becomes the secondary coordinator. Looking back at operation 43 in figure 18 DS_p is the primary coordinator. DS_0 handles stripe unit 0 and therefore, takes the role of the secondary coordinator and DS_1 is a participator.

During the workflow the primary coordinator periodically checks the age of an operation and takes action once a certain threshold is reached. This could be the case when a message is lost and a participator does not perform a certain action. The primary coordinator investigates the status of the timed out operation and identifies the missing response messages. It contacts these servers and, based on the response, decides whether the operation can continue or has to be aborted.

The secondary coordinator in principle acts the same way the primary coordinator does, however, its timeout threshold is longer than the primary coordinator's threshold. Therefore, detecting a timeout at the secondary coordinator is a strong sign that the primary coordinator has failed. It tries to contact the primary coordinator and inherits its role if it cannot be contacted. Furthermore, the secondary coordinator reports the operation result to the client.

4.4.2. Errors In Phase 1

This covers the client's request, the data server's *received*, *perform* and *can-commit* messages. No event that interrupts processing of these operations has any effect on the global state of the stripe. No write operations have been performed. The coordinator must decide whether the operation can be completed or must be aborted. Aborting simply requires freeing up some memory. The client will either be informed by the secondary coordinator, or detect a timeout itself.

4.4.3. Errors In Phase 2

In this phase data has already been written to disk. Therefore, it must be decided whether the operation can be completed or needs to be aborted. Several scenarios have already been described from sections 4.3.5 to 4.3.8.

4.4.4. Errors In Phase 3

All participants performed the operation and the parity server finally forces the kernel level buffer flush (using fsync()). If this succeeds, the operation is performed and the success messages are sent. A failure in this phase could either be a lost committed/success message or a failed primary coordinator.

The secondary coordinator and the participators need to check the timeout interval of the success message, because the primary coordinator is not going to do this. The worst case scenario here is that a participator detects a lost success message and contacts the primary coordinator. This however, already remove all data concerning that operation and has no knowledge about it. Now the participator investigates its state based on the mechanism used after a system crash (refer to section 4.3.5). Should a committed message get lost, the primary coordinator would detect the timeout and investigate the state of the operation. During this process the success message will be resent.

Should the whole primary coordinator have failed, the secondary coordinator detects the timeout and coordinates the operation. The process continues without the parity operation and the failed server needs to be replaced. This has been described in section 4.5.

4.4.5. Flipped Bits

A bit can flip at every electronic component and therefore, must be considered. The metadata of the data object are protected by a 32 bit parity checksum. Since the metadata are only a few bytes, the simple XOR parity is sufficient. Furthermore, the metadata is stored on every participating data object, which adds further redundancy. The user data integrity itself is not handled by the data server. A more sophisticated checksum, such as *cyclic redundancy check (CRC)* or *Fletcher's checksum*[18], could be applied on the user data inside the client library and would be transparent to the data server cluster. Additionally, the client can read the complete stripe and check the parity of the data objects with the provided parity data object.

4.4.6. Interrupted Disk I/O

The data objects are written to disk using fsync() and fclose(). Fsync not only flushes the files data, but also the files metadata. Therefore, the moment the call returns the data is written to disk and the file has been crated by the hosts file system. Depending on the hosts file system, a crash during a write operation might create a file with incomplete data. Ext3, using the *writeback* journaling level, is one of these examples. It only journals the files metadata and writes the data directly to disk. In this mode there is no ordering between writing the metadata to the journal and the data to the disk. A host failure might interrupt the data write operation while the metdata has been journaled successfully. This would mark the files size to be grater than the actually written data. The checksum of my data object's metadata is appended to the file's data (fig. 13) and therefore, guarantees the successful write or an incomplete write would be detected. The probability of undefined memory to accidentally represent the required checksum is $\frac{1}{2^{32}}$ and can be neglected.

4.5. Replacing Data Server

In large clusters, server will fail. K. V. Vishwanath and N. Nagappan [30] discovered that in large scale systems about 8 % of all nodes face a failing component each year. They analyzed the maintenance reports of over 100.000 servers at Microsoft's data centers. Due to Microsoft's hardware policies about 90 % of the servers are less than four years old. Nevertheless, the average number of failures among these servers is 2. Therefore, handling device failures in large scale systems becomes an every day business and must be handled by my system as well. Currently, my architecture only provides single failure protection. Consider the case of a broken network interface card. The data server is unavailable but the service continues to operate using the parity data. However, until the broken network interface card is replaced, the remaining system is vulnerable. It can not compensate another failing server. The mean time to repair (MTTR) must be minimized in order to reduce the window of non-redundant operation. This is very important since failures tend to appear in bursts[9]. For my architecture I analyzed two automatic recovery strategies.

4.5.1. Hot-spare Data Server

A hot-spare server is a data server that is not serving any files yet. It is ready to take over the role and the identity of a failing server. This can be considered to be the server equivalent of a hot-spare disk in RAID systems.

The advantage of this strategy is it's little overhead. The metadata server only assigns a new address for the failed ID and the data server registered at that address, starts to restore the lost data. This simplicity also implies the big disadvantage. The system uses Object RAID and many servers can be used to reconstruct the lost data. Nonetheless, the hot-spare server has to write all the data previously stored on the lost server to disk. Panasas *ActiveStor 12*[5] has

[5]http://www.panasas.com/sites/default/files/uploads/docs/panasas_activestor_

a maximum capacity of 60 TB and a peak write throughput of 1600 MB/sec. Reconstructing 60 TB would take at least 640 minutes and is clearly too much.

4.5.2. Object RAID Recovery

This strategy basically implements the Object RAID recovery strategy introduced by Welch, Unangst, Abbasi, et. al [33]. In contrast to Panasas' strategy, my architecture is going to coordinate the recovery process internally. The metadata server is not involved. Table 1 shows three files and their distribution across the cluster. This is a short version of the actual file layout only showing the assignment to the data server.

InodeNumber	Id_0	Id_1	Id_2	Id_3	Id_4	Id_5	Id_6	Id_7	Id_8
1000	**12**	18	2	45	33	76	88	21	97
1001	43	82	54	67	39	24	**12**	91	89
1002	11	22	33	44	55	66	77	88	**12**

Table 1: File assignment to data server

Let the system run a RAID-4 configuration with a group size of 9. Then the file 1000 would be striped across the data servers 12, 18, 2, 45, 33, 76, 88 and 21 and the parity data would be managed by data server 97. The architecture does support multiple groups, but for this example one is sufficient. Let's assume the data server *12* fails.

It was handling both file 1000 and 1001 and was the coordinator of file 1002. First the system must identify the dead server. This could be detected by a client or another data server during a coordinated operation. Should a client suspect a data server to be unavailable, it informs the parity server of the file it tried to access. Remember the file layout has been retrieved from the metadata server in the first place. In the above example for file 1000 the data server 97 would have been informed. This then checks whether 12 is available and announces it to be dead. This can be done either by a broadcast message or a rumor spreading algorithm. The centralized broadcast would be the most efficient with respect to the number of transferred messages.

Since the recovery process is handled by the data server cluster, each data server checks all its files to see whether it shares a group with data server 12. For very large file systems it would be necessary to maintain a special data structure to perform a fast mapping of $DS_{ID} \rightarrow set(files)$. For billions of files this mapping is going to be too big for an in-memory data structure. In my architecture the parity server of a stripe is responsible for the reconstruction of that stripe. Therefore, a good compromise would be to keep a mapping of 128 InodeNumbers per data server id and start reconstruction on these files. This should keep the server busy while a back ground thread scans the complete file system. The resulting producer-consumer pattern should keep the server busy. 128 InodeNumbers require 1 KB of memory, so that even a mapping of several thousand data server does not stress main memory. Figure 21 shows

`ds_lr_1072.pdf`

the recovery process of the file with InodeNumber 1000.

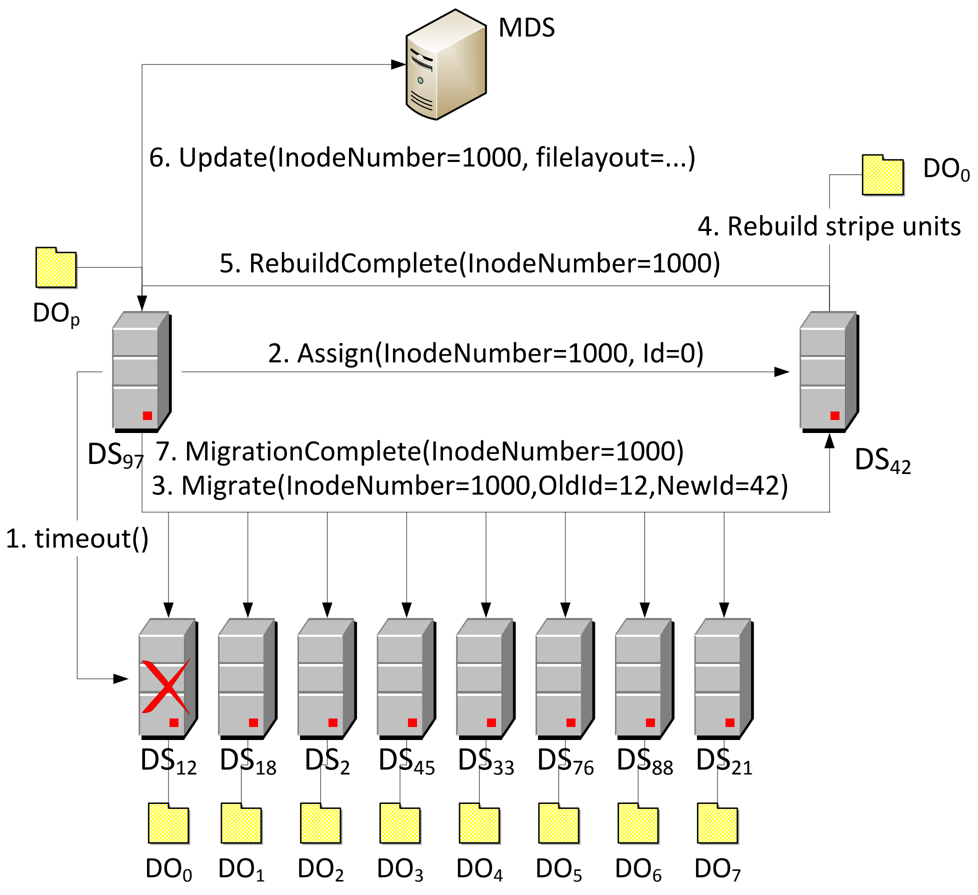
MDS
6. Update(InodeNumber=1000, filelayout=...)
DO$_0$
4. Rebuild stripe units
5. RebuildComplete(InodeNumber=1000)
DO$_p$
2. Assign(InodeNumber=1000, Id=0)
DS$_{97}$
7. MigrationComplete(InodeNumber=1000)
3. Migrate(InodeNumber=1000,OldId=12,NewId=42)
DS$_{42}$
1. timeout()
DS$_{12}$ DS$_{18}$ DS$_2$ DS$_{45}$ DS$_{33}$ DS$_{76}$ DS$_{88}$ DS$_{21}$
DO$_0$ DO$_1$ DO$_2$ DO$_3$ DO$_4$ DO$_5$ DO$_6$ DO$_7$

Fig. 21: Recovery process of file 1000 managed by data server 97

1. During an operation on file 1000, data server 97 has detected a timeout caused by server 12. It identifies 12 to be unavailable and starts the reconstruction process on file 1000.

2. The new data server is informed on its new file and its role in the group.

3. Then all members of the group are informed that the data server 12 is unavailable and will be replaced with id 42. Client write requests are delayed until reconstruction is complete.

4. Based on the provided file layout the new server reads all stripe units and calculates its own.

5. Once all stripe units are rebuild, 42 informs the coordinator.

6. The coordinator updates the file layout at the metadata server.

7. Finally, the coordinator broadcasts the completion of the migration process and the data server start processing delayed client requests.

In case the coordinator itself is unavailable, such as file 1002, the first data server (11 for file 1002) is responsible for the reconstruction. The new data server is determined randomly among the set of data server that are not already listed for the file. For large file systems all data server are potentially responsible to coordinate the reconstruction process of some files. Therefore, choosing the replacing data server randomly should provide the most balanced load to all data server and utilize the cluster's bandwidth best. Once the reconstruction process finished, each data servers load balancing process checks its state and might relinquish participation on some files.

The big advantage of this strategy is the utilization of all available server. The write performance of all servers can be used to reconstruct the lost data objects. No hot-spare server is needed because all the data servers are part of the reconstruction process. Clearly the downside of this approach is the required update of the file layout. This requires a write operation to the metadata server and could become a bottleneck for file systems with lots of small files.

4.6. Workflow Optimizations

The workflow presented in figure 16 has room for optimization. Especially the latency and blocking behavior is discussed in this section.

4.6.1. Full Stripe Write

The cluster has to perform the parity calculation in order to keep the redundant data consistent. This not only creates a higher load on the data server's CPU, but also creates a high network traffic. Since every operation delta has to be sent to the parity server in order to update the parity block, this creates an network traffic overhead of 100 %. At small write operations this is acceptable, but at a full stripe write operation this can be handled in a better way.

At a full stripe write the parity block does not need to be updated, it can be replaced. Figure 22 shows the tailored workflow of the full stripe write. First the client calculates the parity block and sends all data blocks to the appropriate data server. The coordinator manages the operation the same way it does during the normal mode. The difference here is that the new parity block already exists and the data server do not provide the operation delta to the coordinator. The cancommit message is still needed to create the new version vector, but does not carry a large data block.

This operation is still performed lock-less and will not interfere with parallel occurring operations on that stripe. With the help of the version vector, parallel operations are either performed before the full stripe write or after it. Therefore, the stripe's consistency is preserved.

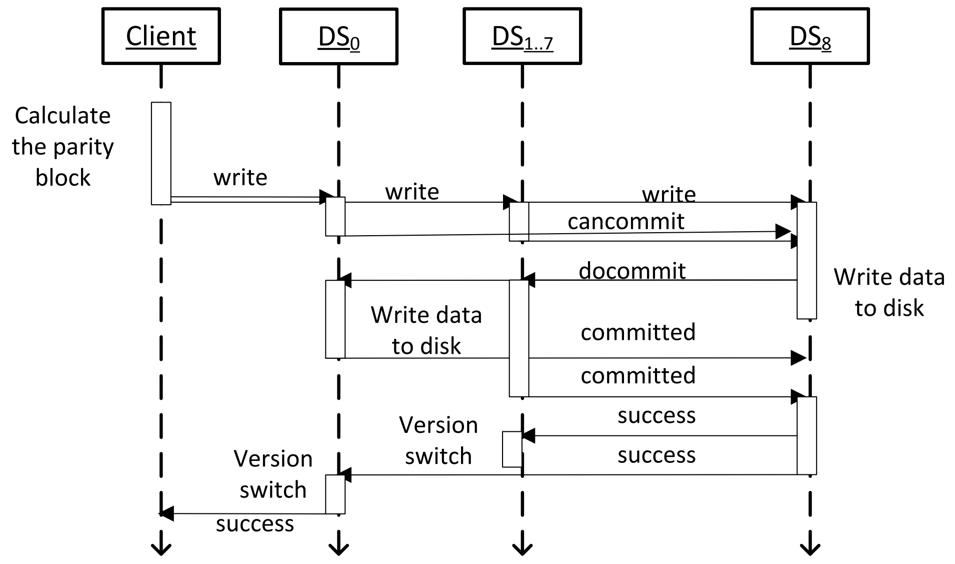

Fig. 22: Full stripe write workflow

4.6.2. Blocking vs. Non-blocking

The workflow described above can be considered a blocking one. The coordinator waits for all data server to finish the current phase until it moves on to the next. This implies that slow data servers delay the workflow and therefore, for a short period, block the faster data servers. A non-blocking version of the workflow will not allow a data server, except for the coordinator, to slow down the operations of other data server.

Initially this architecture was designed and implemented with a blocking workflow. Since my supervisor was interested in a non-blocking approach, I designed the following compromise. The full stripe write stays a blocking process, whereas the partial stripe operations are split into multiple, single stripe unit operations. This is done at the client library and transparent to the data server cluster. The advantage is clearly the reduced latency of the smaller operations, however, the disadvantage is that now every operation creates an update to the parity block. For example an operation covering three data server results in one parity block write for the blocking mode. At the non-blocking mode this operation is split into three operation and performed independently, resulting in three write operations at the parity server.

4.6.3. Non-blocking, Low Latency Mode

The workflow is quite complex and has several phases resulting in some additional messaging latency. In cases where latency is the most important factor of the application, the coordination overhead could be minimized to the following workflow displayed in figure 23.

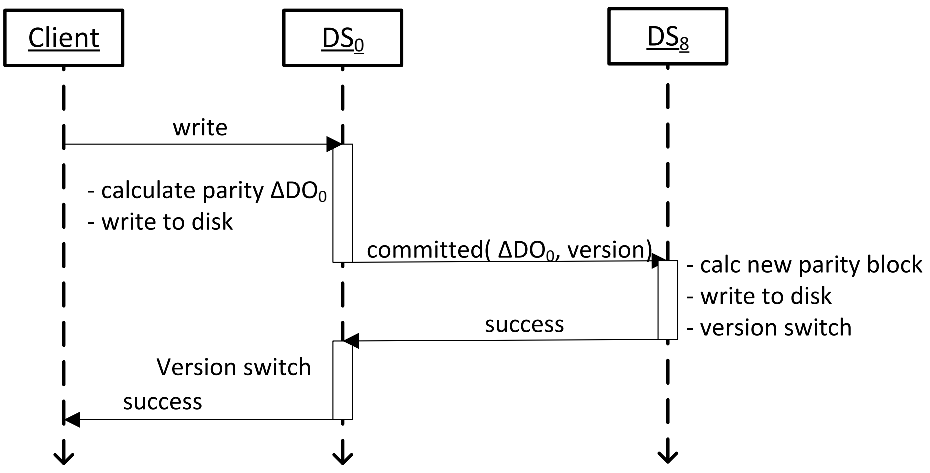

Fig. 23: Non-blocking, low latency mode workflow

Here the data server directly calculates the operation delta and writes the new data block to disk. Then it sends the delta to the coordinator in order to update the parity block. The version of the new data block is part of the committed message and will be written to the version vector of the parity block. On arrival of the committed message, the coordinator considers the data block to be written to permanent storage. Therefore, it can switch to the new version immediately after the new parity block has been written to disk. Then the success message is sent to DS_0 and on arrival of this, it switches to the new version, too. The operation is performed and confirmed to the client.

This low latency version only applies to the non-blocking workflow. There is no way for the coordinator to broadcast the new version vector if more than one participant is involved. Furthermore, I see a problem at an overloaded cluster. DS_0 performs resource intensive operations, such as the parity calculation, disk I/O and the delta transport to the coordinator, without having the guarantee that the coordinator is able to process it within the allowed time frame. Should the operation time out, DS_0 needs to undo the performed operations and therefore, consume even more resources. The first phase of the normal workflow guarantees that the expensive tasks are only performed if all participating data server have the necessary resources. One way to deal with this issue could be the introduction of more priority level. DS_8 would then operate the incoming message sent by DS_0 with the highest priority. However, this would not help in case of a jammed network interface on either of the data server.

4.6.4. Out-of-order Execution And Prefetching

On an idle system prefetching data from disk or out-of-order execution of tasks could reduce the latency of single operations. Figure 24 marks the section in the workflow that have potential for these mechanisms.

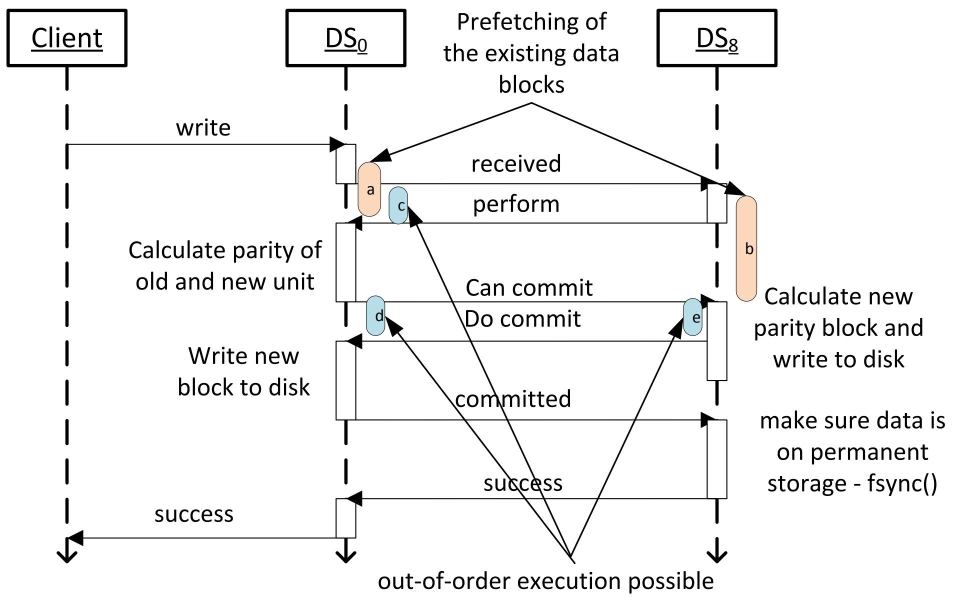
Client
DS_0
DS_8
Prefetching of the existing data blocks
write
received
perform
a
b
c
Calculate parity of old and new unit
Can commit
Do commit
d
e
Calculate new parity block and write to disk
Write new block to disk
committed
make sure data is on permanent storage - fsync()
success
success
out-of-order execution possible

Fig. 24: Out-of-order execution and prefetching potential of the workflow

DS_0 could read the existing data object, which will be needed to calculate the operation delta, while waiting for the perform message of the coordinator (a). The coordinator can do the same on its side (b). At (c/e) the data servers can start to perform the parity calculations. For the coordinator this applies only to operations with more than one participating data server. On arrival of the first cancommit message the coordinator can start to process that data block while waiting for the other messages. At (d) the data server could write the data to disk, even though the docommit message did not arrive yet.

4.6.5. Pipelining

Network traffic is a big issue in distributed file systems, since the transport time of megabyte-scale data is a considerable factor. A data server, receiving a write request, does not process the user data until phase two of the workflow starts. The data necessary to operate phase one (received/perform) is provided inside the client request's head and could be processed while the user data is still on the wire. Depending on the utilization of the network interface and the storage hardware, pipelining could be used to perform phase one of the workflow and reading of the existing data block from disk, while the new data is still being received.

4.7. Loadbalancing

In this section I will describe a sketch for a distributed load balancing mechanism inside the coordinated cluster.

Balancing the load among the data server is a crucial aspect in distributed file systems. In general, a file system should provide low latency for client operations and provide a throughput near to the disk's technical capabilities. Therefore, local file systems try to hide the disk's latency by caching data and buffering write operations. For distributed file systems these techniques are used too, but there are some further issues to address. At local file systems the underlying storage's capacity is constant, whereas in distributed file systems the single disk's usage among the servers vary. Furthermore, the network bandwidth becomes a factor that limits the distributed file systems performance. Especially for read intensive systems there is a big difference compared to local file systems. There, cache hit read operations are not visible to the disk and do not cause extra load. In distributed file systems a cache hit would not cause an extra load to the data server's storage, but still needs to be transferred to the client. In conclusion, each data server part of a distributed file system has some resources that should be managed efficiently:

- the utilization of the data server's storage
- the network bandwidth
- the storage read throughput
- the storage write throughput
- the systems request latency

The next sub-sections will explain why these attributes are important to the cluster's performance, how the current state of the data servers can be detected and what operations need to be performed to improve the cluster's status.

4.7.1. Storage Utilization

The storage utilization is obviously an important factor for a data server. Is it too high, the data server can not host new files and therefore, reduces the cluster's write performance. Looking at the read performance a burst of client read requests is more likely to saturate that data server's network connection, while a low utilized data server's connection might be idle.

Since my data server use the host's file system to store the data, the storage utilization is determined by a simple system call. Balancing the utilization among the data server is done by moving from a high utilized data server to a low one. The decision, which file is moved, depends on all the mentioned criteria. A very simple solution would be to move a cold file from a highly utilized data server to another data server with plenty of free capacity. In my architecture the creation time stamp and the version vector of each stripe of a file can be used to identify cold files. Since the version vector is incremented each time the stripe unit is updated, it counts the number of write operations to a stripe unit. Along with the creation time stamp, these attributes are sufficient in order to determine the file's write frequency. Although in POSIX file systems there is a access time stamp that represents the time of the last read operation, for performance reasons this is not updated in my cluster. In NFS 4.1[22] updating the access time stamp is optional. Therefore, there must be another way to identify read intensive files. A low overhead solution would be to check whether a file is cached and consider this to be a hot file and a cache miss to be a cold one. More sophisticated solutions might have a negative impact on the clients request latency and need to be evaluated in a future version of the cluster.

4.7.2. Network Bandwidth

The network bandwidth is extremely important in distributed systems. It directly affects the read and write throughput of the data server and the request latency of a client request. There are several on board tools that monitor the system's hardware and could be used to determine the network interface utilization.

Should a server detect a high network utilization over a certain period, it might consider reducing the number of read intensive files. As mentioned in the previous section, a read intensive file is most likely a cached one with a low update frequency. In my architecture coordinating a stripe is causing a high network load and therefore, migrating write intensive files that are coordinated by the data server would also reduce the network load.

4.7.3. Read Throughput

The read throughput is limited either by the storage read performance or, in case of many cache hits, by the data server's network connection. Basically the same action apply that are performed to improve the network bandwidth. Should the system suffer lots of disk read operations, the capacity misses could indicate an unsuitable access pattern.

4.7.4. Write Throughput

The best way to reduce the write operation load on a data server is to migrate write intensive files. These can be identified by their version index. Furthermore, parity data objects usually suffer high write rates.

4.7.5. Request Latency

A high request latency usually indicates that at least one resource of the data server is facing a high load. In my architecture the round-trip time (RTT) is a good measure to determine the load of the other data servers. The coordinator of an client operation can use the RTT of the *docommit-committed* cycle and estimate the load of that data server. In a sufficiently large file system, every data server acts as a coordinator for every other data server on some files. Therefore, this measurement should provide a good basis to evaluate their load and identify suitable candidates to migrate responsibilities.

4.8. Implementation Details

The cluster has been designed and implemented from scratch. Therefore, the implemented parts must be considered a functional prototype. This section is going to present some insights into the implementation status and the used techniques.

4.8.1. Implementation Environment

The data server runs as a linux user space process and is implemented in C++. Two different networking frameworks are used. The $\oslash MQ$[6] framework is used to interact with the metadata server. This framework is specifically designed to handle lots of small messages and therefore, is a good choice for the metadata server communication. However, it performs poorly for messages of several KB and is unsuitable for the data server's I/O operations. The *boost::asio*[7] framework handles the cluster communication and the storage protocol.

4.8.2. Metadata Server

As already mentioned the MDS is based on the work of our project group and has been extended to fit my requirements. I implemented the ClientSession-Management, a byte-range lock manager and the file layout handling. The BRL is not implemented to provide a fair scheduling of client requests. During the evaluation, the BRL mechanism is used to emulate the stripe lock mode and is performance critical. Therefore, I chose to implement a fast, but unfair busy waiting mechanism, rather than a fair, but potentially slow one. In a ready to deploy system this mechanism needs to be reimplemented, but for now it provides good benchmarking results.

Currently, the system supports only one metadata server. This is an issue introduced by our MDS. It is designed to serve a NFS workflow, which handles client requests based on the mounted file system tree structure. Without going into further details here, our MDS cluster does not provide a look up of $InodeNumber \rightarrow responsibleMDS_{id}$. Therefore, a data server, which has no knowledge about the file systems tree structure, cannot identify the currently responsible metadata server for a file system object. Nevertheless, at the current proof-of-concept status of a coordinated data server cluster, a single MDS is sufficient. The cluster configuration is done at the MDS. The *Libssh2* [8] library is used to distribute the data server configuration files and start the data server processes at the defined hosts.

[6]www.zeromq.org
[7]www.boost.org/libs/asio
[8]www.libssh2.org

4.8.3. Client Library

At the client library the interaction with the metadata server and the data server has been implemented. The client supports the metadata operations and data server I/O operations. The Filesystem in Userspace (FUSE) support has not been implemented yet. This thesis is supposed to investigate the potential of a coordinated data server cluster and focuses on a very specific workflow. I found it very hard to generate this workflow with the known file system benchmarks and therefore, developed my own benchmarking tool. This directly interacts with my client's library and does not rely on the FUSE API. For further details on the benchmarking environment refer to section 5.2.1.

4.8.4. Data Server

The data server processes requests asynchronously. Figure 25 shows the queueing model of the data server core, the storage interface and the cluster interface. The workflow can best be explained on a simple write request example:

The client established a connection to the data server and sends a write request message. The data server's storage interface receives the request and creates a task data structure. This structure identifies the type of the operation and stores all necessary attributes. A pointer to the task is then enqueued into the appropriate queue of the storage interface. The data server's dispatcher pops the pointer and determines the type of the task. Based on that type the *OperationManager* creates a new operation structure and starts to process the new write request. The operation structure keeps track of the operation status and all the involved attributes. The OperationManager creates a task that will send the *received* message to the coordinator and enqueues it. The dispatcher forwards the task to the cluster interface's main queue. The cluster interface identifies the task's type and creates the necessary messages. Especially, the coordinator often broadcasts messages to all participants of an operation. Therefore, an extra queue is implemented in order to send several messages in parallel.

The priority queue is designed to process open operations first. Initially, the client sends a request via the storage interface. The queue's callback function puts the task, generated by the request, into a queue with a lower priority than the cluster interface. Therefore, the messages received by the coordinator during the coordination workflow are preferred over new incoming client requests. On an overloaded system this reduces the timeout rate of active operations. Some management tasks, such as the garbage collection, are performed with a lower priority, whereas other tasks, such as the timeout monitoring, is performed with highest priority. Currently the main queue is processed by 10 threads and the network interfaces operate with 5 threads.

The prototype is implemented to provide a proof-of-concept. Therefore, several features that are not critical, with respect to the evaluation, have not been implemented yet. The recovery process of a failed data server has not been implemented. First of all the recovery process can not be evaluated independently. Since the filelayout update is required, the metadata server would be

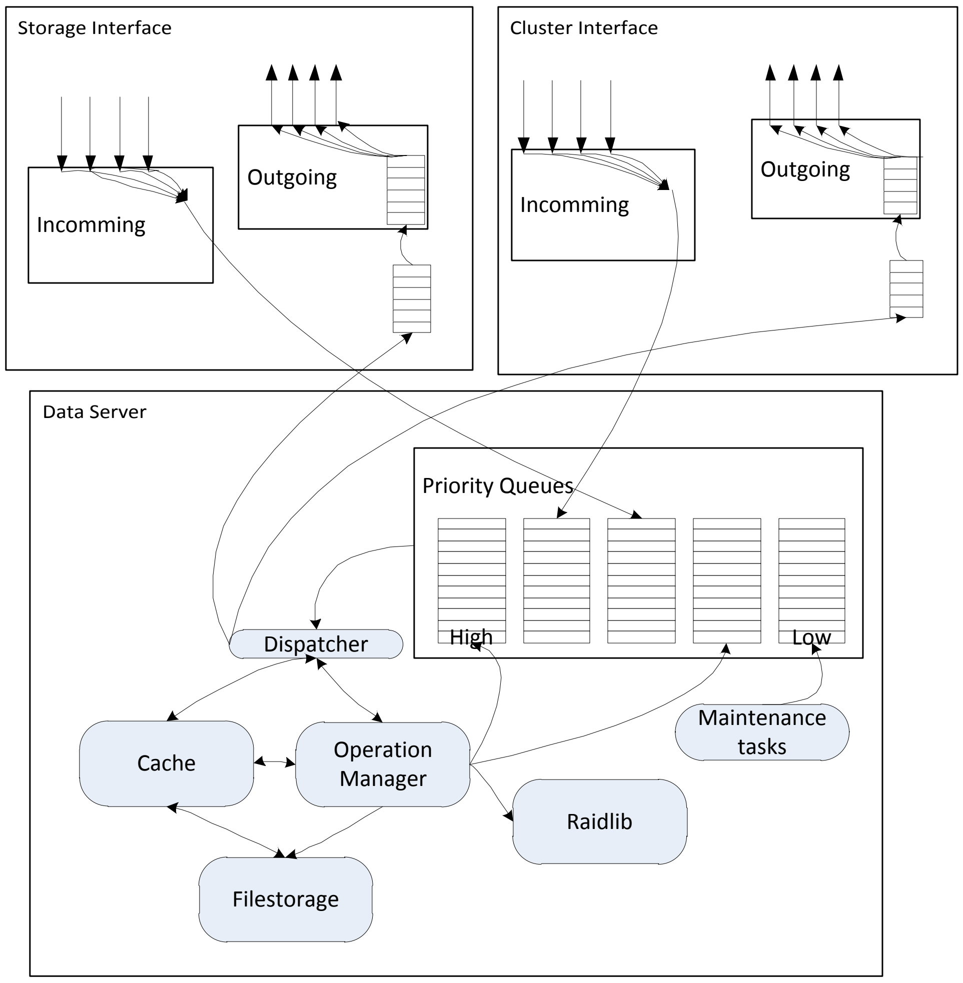

Fig. 25: Implemented queueing model

part of the evaluation, which makes it difficult to argue on the actual cluster performance. From a scientific view, the interesting part of the recovery process is the interaction with the distributed load balancing process. Evaluating the recovery process on a static cluster is basically a combination of read and write operations. These are going to be evaluated independently by my benchmarker.

Several error cases, such as disk errors or failed memory allocations, are not handled and would deadlock an operation. The internal cache currently does not have a replacement strategy. For a 1 MB data object, the cache can manage about 1024 objects per gigabyte (GB) of RAM. A good replacement strategy will be crucial in a real live system, but would not have any effect on my current benchmarks. The evaluation is going to show that the CPU power is not the critical part of the system.

4.8.5. Data Efficiency

Each stored data object has some metadata attached, which causes an overhead. Table 2 shows the metadata attributes and their size. The version vector size depends on the system's configuration. For a single parity RAID a configuration of more than 24 data servers sharing a parity object is unreasonable and 100 Bytes is a safe upper bound for the version vector's size. Therefore, the upper bound of the whole metadata can be considered to be 400 Bytes.

size in Bytes	Attribute
8	client operation id
8	operation offset
8	operation length
256	file layout size
4n	version vector size with n being the group size (9 for a 8+1 RAID 4)
8	data block length
8	metadata checksum
296 + 4n	sum

Table 2: Data objects metadata size

Considering a 1 MB stripe unit size, this produces a negligible overhead. Currently, the metadata causes the allocation of an extra block on the disk and therefore, occupies a full block of 4 kilobyte (KB). Reducing the stripe unit size by the metadata size would prevent this. During the evaluation I do operation with full 1 MB data blocks and accept the extra block for the metadata.

size in Byte	Attribute
38	message head
12	protocol head
294	operation head
x	variable size message part
344 + x	sum

Table 3: Messaging overhead

The messaging produces some overhead, too. Table 3 displays the message structure. The message head contains some information needed by the network layer to identify the sending data server and the size of the variable length part of the message. The protocol head provides some protocol specific information and the operation head provides all information necessary to identify the client operation that the message belongs to. These could have been encoded into the variable length part of each message or into the protocol specific head, but due to the relatively small overhead of 294 Bytes, I consider this a micro optimization. At this stage of the prototype this overhead is acceptable.

5. Evaluation

During this section the implemented system is going to be evaluated. In section 5.1 the evaluated system environment and the data server configuration is presented. The following section 5.2 explains the benchmarking tool and the performed benchmarks in detail. Several system component benchmarks have been performed. These are presented in section 5.3.

After that the actual results are presented and interpreted. Section 5.4 compares the throughput of the stripe-lock mode benchmark with the coordinated cluster mode and section 5.5 presents the component benchmarks. Section 5.6 and 5.7 analyze the stripe-lock mode and the coordinated cluster mode in detail. The results of the full-stripe write operation is evaluated in section 5.8 and the effects of different stripe unit sizes on the coordinated cluster workflow are presented in section 5.9

5.1. System Environment

The test setup is displayed in figure 26. The data servers form a RAID 4 array where DS_0 to DS_7 store the files user data and DS_8 the parity data. At this scale one metadata server is sufficient and runs on a separate physical host. The focus of this thesis lies on conflicting client operations. This means that there will be no massive cluster throughput benchmarking. The evaluation will concentrate on a single stripe and is designed to evaluate the performance of the parity object of that stripe. Therefore, a single client host triggering the operations is sufficient.

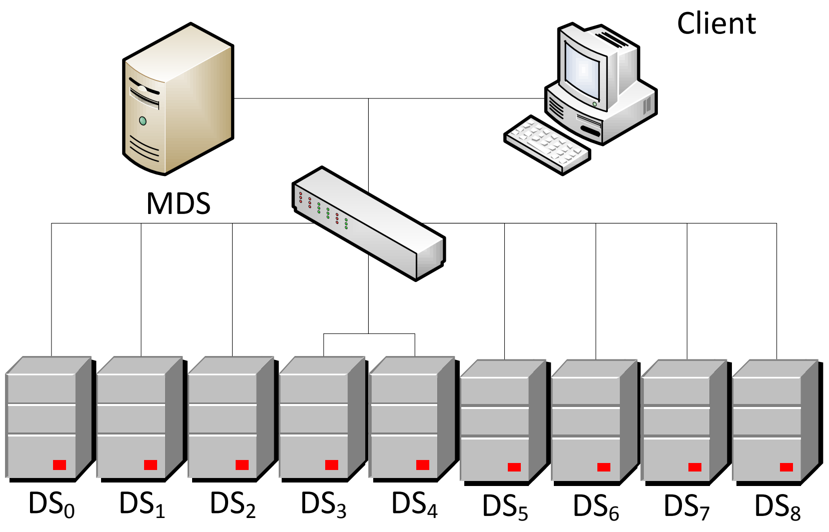

Fig. 26: Benchmarking environment system setup

5.1.1. Grid5000

The servers are part of the Grid5000 [14] network. It connects the sites Bordeaux, Grenoble, Lille, Luxembourg, Lyon, Nancy, Orsay, Reims, Rennes, Sophia-Antipolis and Toulouse. Each of these sites host several clusters with different hardware configurations. My benchmarks were executed at the *Suno* and *Helios* clusters at the site Sophia-Antipolis, the *Griffon* cluster at the site *Nancy* and the *Granduc* cluster at the site Luxembourg. The hardware specifications of these clusters are presented below.

Suno cluster: 45 Dell R410 nodes[9]	
CPU:	Intel Xeon E5520
	2.26 GHz, 8 MB, 1066 MHz FSB
	2 CPUs per node, 4 cores per CPU
Memory:	32 GB
Network:	Gigabit Ethernet (Broadcom NetXtremeII BCM5716)
	driver bnx2
Storage:	2x300 GB / SAS/RAID-0
	Driver: megaraid sas
Misc:	BIOS settings: Cstate=on, Power Management=OS, Turbo=Off,
	Hyperthreading=Off

Table 4: Suno cluster details

Helios cluster: 56 Sun Fire X4100 nodes[10]	
CPU:	AMD Opteron 275
	2.2 GHz, 1 MB, 400 MHz FSB
	2 CPUs per node, 2 cores per CPU
Memory:	4 GB
Network:	Myrinet-2000 (M3F-PCIXF-2)
	driver e1000
Storage:	2x73 GB / SAS/RAID-0
	Driver: mptsas

Table 5: Helios cluster details

5.1.2. Software Environment

The server run a 64-bit Debian squeeze with kernel version 2.6.32-41. The host's file system is ext3. The data server run as a user mode process and use the memory manager *tcmalloc* from Googles performance-tools[13]. This provides the fastest response times of seven tested memory managers[8].

[13]http://code.google.com/p/gperftools

Griffon cluster: 92 Carri System CS-5393B nodes[11]	
CPU:	Intel Xeon L5420
	2.5 GHz, 12 MB, 1333 MHz FSB
	2 CPUs per node, 4 cores per CPU
Memory:	16 GB
Network:	Infiniband-G20 (MHGH29-XTC)
	driver e1000e
Storage:	320 GB SATA-II
	Driver: ata_piix

Table 6: Griffon cluster details

Granduc cluster: 22 Dell PowerEdge 1950ES nodes[12]	
CPU:	Intel Quad-Core Xeon L5335
	2 GHz / 2x4MB 1333 MHz FSB
	2 CPUs per node, 4 Cores per CPU
Memory:	16 GB FB 667MHz Memory (8x2GB dual rank DIMMs)
Network:	Intel 10 GbE
	Driver: ixgbe
Storage:	146 GB / SAS(10,000rpm) 2.5in

Table 7: Granduc cluster details

5.2. Benchmarking

The question to be answered by this thesis is whether a coordinated data server cluster can perform access control measures and avoid the need of global stripe-locks, while increasing the performance of conflicting write operations. I described a consistency model and a workflow that will maintain a consistent file state. This section describes the used benchmarking tool and the applied benchmarks.

5.2.1. Benchmarker

As explained in section 4.8, FUSE support has not been implemented yet. This also implies that the system is evaluated using my own benchmarking tool which directly accesses my client's library. I examined some existing file system benchmarking tools, such as *bonnie++*[14], *iozone*[15] and *dbench*[16]. Since my thesis handles a very specific type of access pattern, none of these is suitable. I intend to perform several benchmarks that cover only single parts of an client operation, such as the lock management or the message passing latency. Even if I had found a suitable benchmarking tool, these tests would require the implementation of my own benchmarking tool.

My benchmarker has been implemented in C++ and directly accesses the

[14]http://www.coker.com.au/bonnie++/
[15]http://www.iozone.org/
[16]http://dbench.samba.org/

client library's API. This provides the needed flexibility to test single functions like *acquireLock()* and the needed preciseness to measure each operation's latency.

5.2.2. Stripe-lock Mode

The stripe-lock mode represents a parallel Network File System (pNFS) based distributed file system. The performance of my workflow is going to be compared to this one. This comparison can only be valid if there are no hidden bottlenecks or badly implemented functions within this benchmark. So lets first have a look at the detailed workflow of the benchmark and then discuss some assumptions I made.

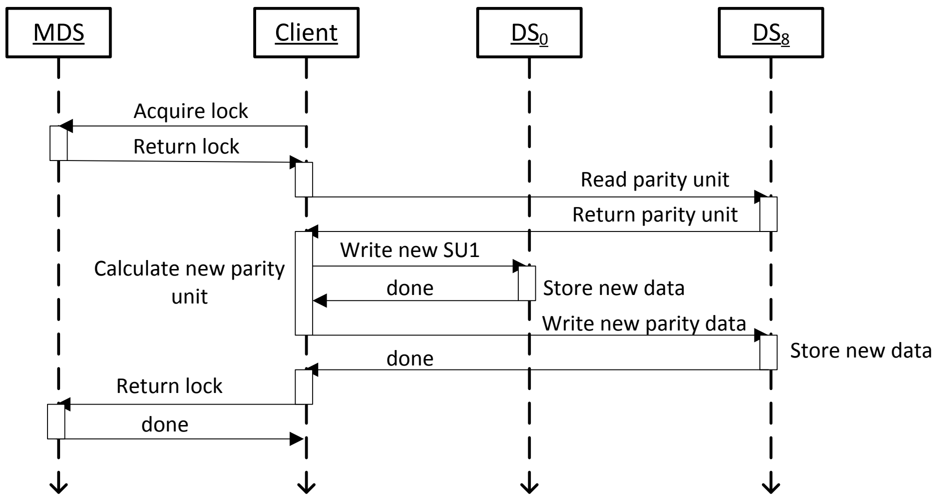

Fig. 27: Stripe-lock mode benchmark workflow

Figure 27 displays the workflow of a client write process accessing a single stripe unit. The complete benchmark will cover eight runs with up to eight competing write operations. First, the client acquires the stripe-lock from the metadata server. Client and MDS communicate synchronously. The MDS was developed to run in another environment and not to directly interact with clients. For large scale systems synchronous communication would not scale, however, for this evaluation I implemented the lock management to do busy waiting. This way there is a very small delay between release from $client_A$ and acquire by $client_B$. More details on the specific lock management benchmark can be found in section 5.3. Once the client received the lock, it has to build the new parity data object

$$DO_p^{i+1} = DO_p^i \otimes DO_0^k \otimes DO_0^{k+1}.$$

First, the client reads the old block DO_p^i from the responsible data server, which is DS_8 in the above diagram. At that data server the data object is cached and therefore, no disk read operations is necessary. I made this assumption because I emulate an interactive system with plenty of operations on that data object. In real world systems a cache hit would be very likely. After the client read the old parity object, the new one is calculated. The old data object DO_0^k is cached by the client and will not be read from DS_0. This implies that the operation

$$\Delta DO_0 = DO_0^k \otimes DO_0^{k+1}$$

can be performed before acquiring the stripe-lock and therefore, is omitted from this benchmark. That operation comes build-in at my coordinated cluster, so here I assume the best case for the stripe-lock workflow. The objects DO_0^{k+1} and DO_p^{i+i} are then sent to DS_0 and DS_8 in parallel. The data servers do not communicate with each other. The data is written to disk immediately with the sync flag set and a response is sent. After both servers confirmed the write operations, the lock is returned and the workflow is complete.

For this evaluation I rely on correctly behaving clients. Clients only write data if they received the necessary lock. Without this assumption, a sophisticated distributed lock management would have to be implemented. Since my coordinated cluster does not need stripe-locks, comparing the results would be difficult and a badly implemented lock management could easily corrupt the stripe-lock benchmark's results. Therefore, omitting the lock management on the data server can be considered a constant time verification algorithm.

The lack of a precise specification of the access control to redundant data in the pNFS standard made it difficult to define a valid evaluation workflow. Since Panasas uses a stripe-lock mechanism, my stripe-lock algorithm should be a reasonable approach, too. Furthermore, I made the uncertain assumptions, such as caching and lock management, in favor off the stripe-lock mode. Therefore, the benchmark should produce good results and a valid basis for comparison.

The measured attributes are the performed operations per second and the latency per client operation. Given a predefined stripe unit size s, the throughput on the parity object can be calculated as follows: $T = s \times \#clients \times Ops/sec$.

Both, for throughput and latency on every 50 operations the average is calculated. For a thousand individual operations this produces twenty results values. On these values the average and the confidence intervals are calculated.

5.2.3. Coordinated Cluster Mode

This benchmark evaluates the performance of the coordinated cluster mode. Figure 28 displays the workflow of a single client operation.

In contrast to the stripe-lock mode, neither the metadata server is involved, nor the client has to care about the redundant data. It directly sends the new stripe unit to the appropriate data server. This implies that it is unimportant

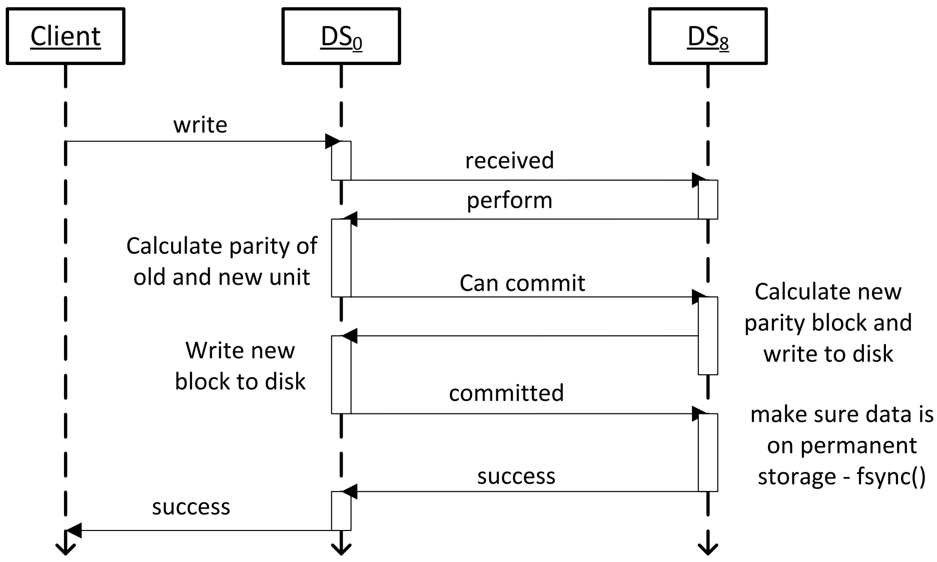
Client
DS0
DS8
write
received
perform
Calculate parity of old and new unit
Can commit
Calculate new parity block and write to disk
Write new block to disk
committed
make sure data is on permanent storage - fsync()
success
success

Fig. 28: Coordinated cluster mode benchmark workflow

whether the client cached the old data object. With the same arguments, as explained for the stripe-lock mode, I do assume the data server caches the latest versions of their data units.

Once DS_0 received the client's request, it sends a message to the parity server DS_8. This checks whether it is the coordinator of this operation and acknowledges it. DS_0 then calculates

$$\Delta DO_0 = DO_0^k \otimes DO_0^{k+1}$$

and sends ΔDO_0 to the parity server (cancommit). The coordinator returns a docommit to DS_0 and calculates the new parity object

$$DO_p^{i+1} = DO_p^i \otimes \Delta DO_0.$$

This is then written to disk, without forcing the system to flush the data to disk. DS_0 writes the new data object to disk with the sync flag set. This guarantees that the data is written to permanent storage when DS_0 sends the committed message to the coordinator. Now the coordinator flushes its data too. Therefore, the data is on permanent storage when the success message is returned. DS_0 forwards the message to the client and the operation is complete.

This operation will be performed with up to eight clients, writing to different stripe units (DO_0, ..., DO_7), which are sharing the parity object on DS_8. The measured attributes are the same as for the stripe-lock mode benchmark.

5.2.4. Full Stripe Write

The coordinated full stripe mode writes a stripe without acquiring a stripe-lock. Figure 29 shows the evaluated workflow.

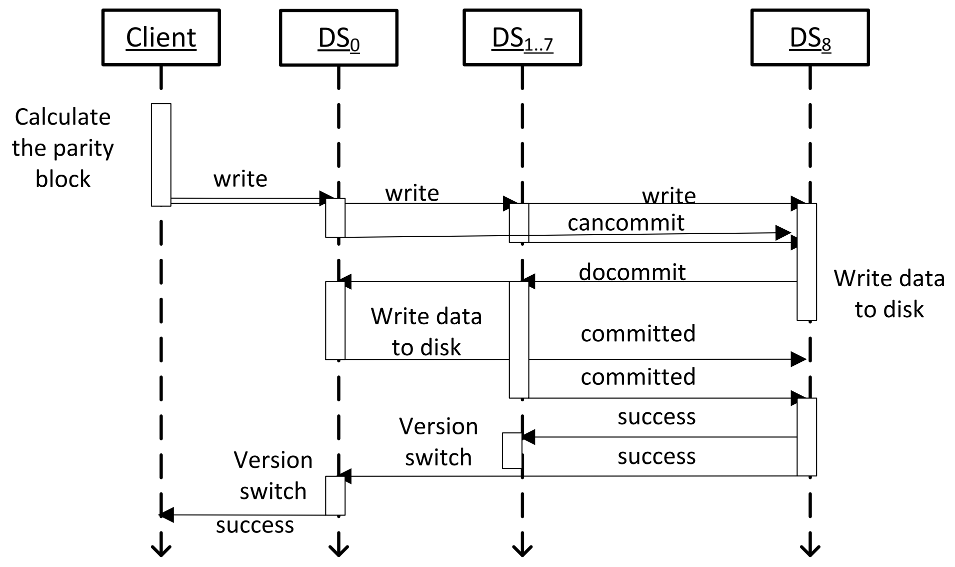

Fig. 29: Coordinated cluster full stripe mode benchmark workflow

First, the client calculates the parity of the new objects its about to write. Then it sends the new objects to the appropriate data server. The data server identify the coordinator of the operation and send a cancommit message. The coordinator gathers the messages from all data servers and returns docommit. Every server writes its object to disk and sends committed to the coordinator. After the coordinator wrote its block to disk and received all committed messages, it returns success to the data server. The secondary coordinator forwards the performed message to the client and the operation is complete.

In contrast to the single stripe unit operation, the coordinator needs to manage the complete stripe group for the same operation, rather than every participant independently. This means that the data server have to wait for the slowest individual during the coordination.

The measured attributes are again are ops/sec and the latency of an operation. The confidence interval is calculated the same way it is for the stripe-lock benchmark.

5.3. System Component Benchmarks

The above explained benchmarks compare the performance of the two systems. However, for complex architectures, such as distributed file systems, these benchmarks are not sufficient. To be able to reason about the advantages and disadvantages of the different workflows, detailed knowledge about the performance of the individual components is needed. The following benchmarks are used to measure individual components of the system. These are both hardware components, such as the CPU, and software components, such as the lock management.

5.3.1. CPU Benchmark

This benchmark performs the parity calculation as it would occur during the normal workflow. The parity of two 4 MB blocks is calculated iteratively and the throughput in GB/sec is measured. The inverse of the throughput then represents the latency per operation.

5.3.2. Disk I/O Benchmark

This benchmark iteratively creates 1 MB sized files and flushes the buffer using the fsync syscall. With a growing number of parallel working threads, this benchmark emulates the disk access pattern of the data server during parallel write operations. The throughput and the latency is measured. For each test run the variance is calculated based on 250 operations per working thread. I reduced the number of operations from 1.000 to 250 because of the tight schedule on the Grid5000 cluster.

This benchmark does not evaluate the streaming throughput of the disks. Scheduling the randomly incoming read and write operations in order to optimize the disk's throughput is a general problem in I/O intensive systems. Both the disk controller and the operating system do take certain actions to improve throughput, however, my architecture overrides these mechanisms by forcing a flush for every operation. A solid state disk (SSD) could be used to buffer write operations and schedule the write back more efficiently. I consider this a future task and therefore, evaluate my data server against the disk's throughput under the given access pattern.

5.3.3. Network Benchmark

Nuttcp[17] is a network benchmarking tool and used to measure the bandwidth of the cluster network. The metadata server starts the nuttcp server and the benchmarker will perform the client operation and determine the bandwidth. It does not evaluate the latency of small messages or the transport time of a larger data block, but evaluates the streaming throughput of the network.

5.3.4. Lock Round Trip Time

This benchmark measures the round trip time of the metadata server's lock management. A growing number of clients iteratively acquire and release the same lock. The result is the latency added by the lock management and measured in milliseconds.

[17]http://www.lcp.nrl.navy.mil/nuttcp

5.3.5. Data Server Messaging Latency

This benchmark issues a message ping pong between two data servers. This benchmark measures the latency of a small message sent over the cluster or storage protocol. The time spent sending a thousand messages between two data servers is measured and the latency of a single message is calculated. This indicates the latency added by the coordination process' messaging.

5.3.6. Stripe Unit Read latency

In general the read performance of a file system is a crucial feature. The focus of my thesis lies on conflicting client write operations, therefore, performing large scale read benchmarks at this point has little scientific value. Nevertheless, the latency of a single read operation is very interesting. This benchmark measures the time the client needs to read data from a server's cache. This directly indicates the latency of the read operation of the parity block during the stripe-lock workflow. Furthermore, if we subtract the time taken by the client request message, we get the transfer time of a large data block.

5.4. Stripe-lock Mode vs. Coordinated Cluster Mode

This section shows and compares the results of the stripe-lock mode and the coordinated cluster mode benchmarks. This comparison will show how much the coordination used to guarantee consistency increases the operation's latency and how the cluster scales for conflicting client operations.

5.4.1. Sophia Results At Suno

The following results were measured at the cluster *Suno* at the Grid5000 site *Sophia*. The stripe unit size was set to 1.0 MB.

Figure 30 visualizes both the stripe-lock and the coordinated mode results. At the Y-axis the operations per second are displayed. Since the stripe unit size is 1.0 MB, this correlates with the throughput in MB/sec. The X-axis represents the number of parallel executed client operations. In order to interpret the results it is important to identify the performance of the network connection and the storage subsystem. Therefore, the yellow graph represents the 1 Gbit Ethernet bandwidth of 112,2375 MB/sec. It has been evaluated with the Nuttcp tool. The detailed testlog of the nuttcp benchmark can be found in table 22 in the appendix. The purple graph represents the bandwidth of the two disk SAS RAID-0 storage of the parity server. Data was measured with the disk I/O benchmark described in section 5.3.2.

The variance seem to be induced by the RAID-controllers caching. The sustained performance for several parallel writing processes is about 150 MB/sec. The results for the 5, 6 ,7 and 8 parallel write operations show a mild penalty with a growing degree of parallelism. More significant is the comparison to the network bandwidth. This shows that at the *Suno* cluster the network tends to

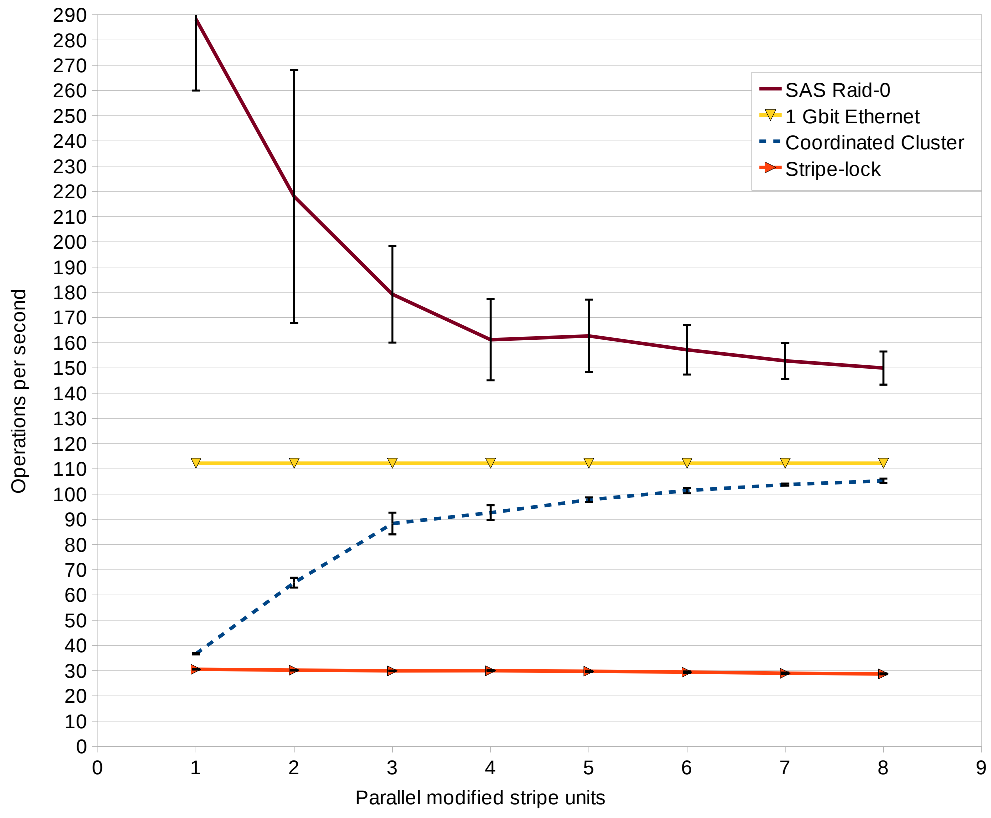

Fig. 30: Write operations per second on a single stripe

be the bottleneck.

The performance of the stripe-lock mode benchmark stays constant. Adding more operations in parallel does not increase throughput. Clearly the stripe-lock prevents parallel execution of the operations. In contrast to this, the coordinated cluster scales well and seems to saturate the network connection. For eight parallel client operations the system handles about 105 ops/sec. The stripe unit size is 1 MB and therefore, the parity server processes about 105 MB/sec on a single stripe's parity object. This indicates that the network connection is the limiting factor for this cluster.

Table 8 shows the results of the stripe-lock mode benchmark. The number of parallel operations can be seen in the first column. The next display the performed operations per second on the tested stripe and the standard deviation σ. Column four displays the average latency of all performed operations for the specific test run and column five its standard deviation. For each test and each participating stripe unit 1.000 operations were executed and the average performance has been calculated.

For a single client operation the stripe-lock mode is able to perform about 30,5 operations per second. Adding more operations in parallel does not im-

#-Clients	Ops/sec		Latency in ms	
	avg	σ in %	avg	σ in %
1	30,541	$\pm 0,3$	32,897	$\pm 0,6$
2	30,197	$\pm 0,4$	66,631	$\pm 1,3$
3	29,915	$\pm 0,4$	99,854	$\pm 3,3$
4	29,975	$\pm 0,7$	133,894	$\pm 2,8$
5	29,768	$\pm 0,7$	167,184	$\pm 6,4$
6	29,417	$\pm 1,0$	204,461	$\pm 5,3$
7	28,968	$\pm 1,0$	239,733	$\pm 6,9$
8	28,717	$\pm 0,6$	275,937	$\pm 7,1$

Table 8: stripe-lock mode benchmark results on a RAID 4 8+1 System

prove the systems performance which drops down to around 28,7 ops/sec. This slight decline is caused by the byte-range lock management system. I did not implement a fair scheduling of the lock requests, but did implement a mechanism that prevents starvation. This mechanism seems to be responsible for the reduced throughput. Furthermore, the unfair scheduling causes a high variance in latency of up to 7,1 % for the 8 client run. Nevertheless, the stripe-lock prohibits a performance improvement for parallel operations.

In contrast to this the coordinated cluster approach does not lock the whole stripe. These benchmark results are shown is table 9.

#-Clients	Ops/sec		Latency in ms	
	avg	σ in %	avg	σ in %
1	36,678	$\pm 0,8$	27,197	$\pm 2,8$
2	65,276	$\pm 2,9$	30,300	$\pm 12,3$
3	87,524	$\pm 4,9$	33,001	$\pm 8,6$
4	92,039	$\pm 3,2$	42,499	$\pm 8,2$
5	97,589	$\pm 0,9$	51,083	$\pm 5,6$
6	101,542	$\pm 1,1$	59,862	$\pm 5,1$
7	103,778	$\pm 0,4$	66,817	$\pm 5,2$
8	105,088	$\pm 0,9$	75,626	$\pm 5,0$

Table 9: Coordinated cluster benchmark results on a RAID 4 8+1 System

Here we can see that adding a second operation in parallel almost doubles the system's throughput. Starting with about 36,7 ops/sec for a single client, system performance increases to 65,3 ops/sec for two parallel operations and to 87,5 ops/sec with three clients. The average latency of the single client operation is 27,197 ms and is even lower than the latency of the stripe-lock mode.

5.4.2. Sophia Results at Helios

The Helios cluster has a Myrinet-2000 network interface operating at 235,231 MB/sec, but only a 2x73 GB RAID-0 storage capable of about 41 MB/sec. In contrast to the *Suno* cluster, the storage is going to be the bottleneck. Figure 31 shows the results of the coordinated cluster and the stripe-lock mode.

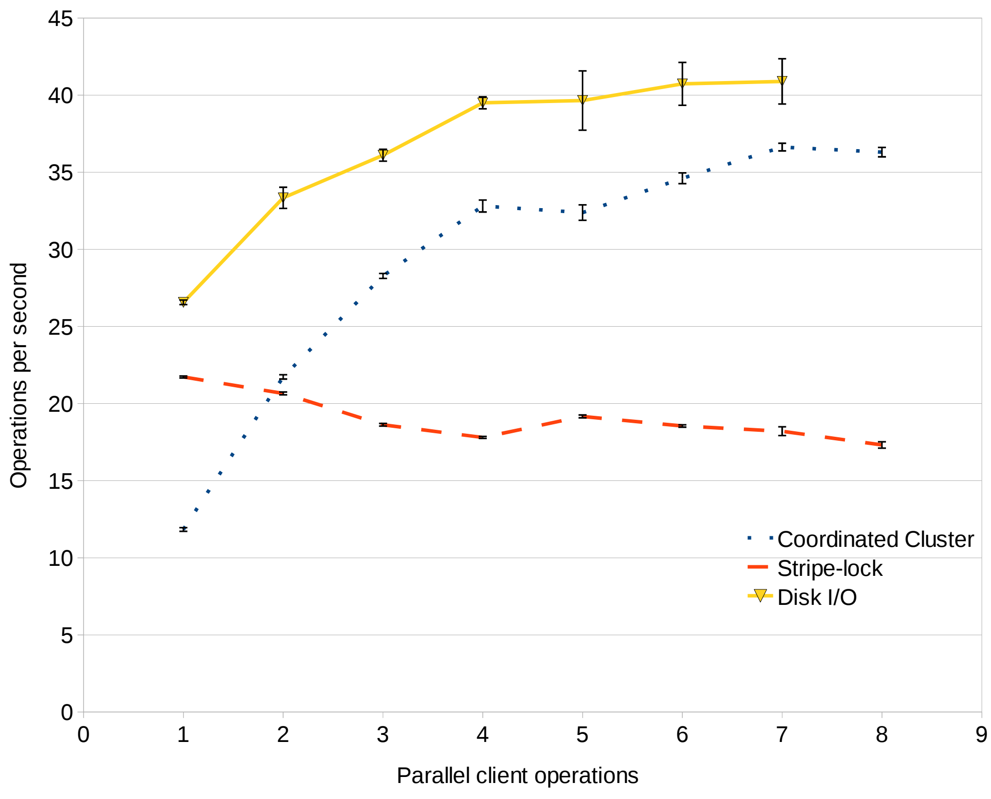

Fig. 31: stripe-lock mode vs. coordinated cluster mode over Myrinet-2000

Clearly the stripe-lock mode performs better for single client operations. This indicates that the stripe-lock mode suffers less penalty from the slow disk and gains more performance from the faster network connection. For two and more conflicting operations the coordinated cluster performs better than the stripe-lock mode. However, the bad performance of the single client operations shows, that the disk I/O performance of the data servers storage layer needs to be improved.

The detailed results can be found in table 23, 24 and 25.

5.4.3. Nancy results

The following benchmarks were performed at the *Griffon* cluster at the site *Nancy*.

Figure 32 shows the results of the griffon benchmark and table 10 lists the exact values. The cluster's storage subsystem is a 320 GB Hitachi SATA-II

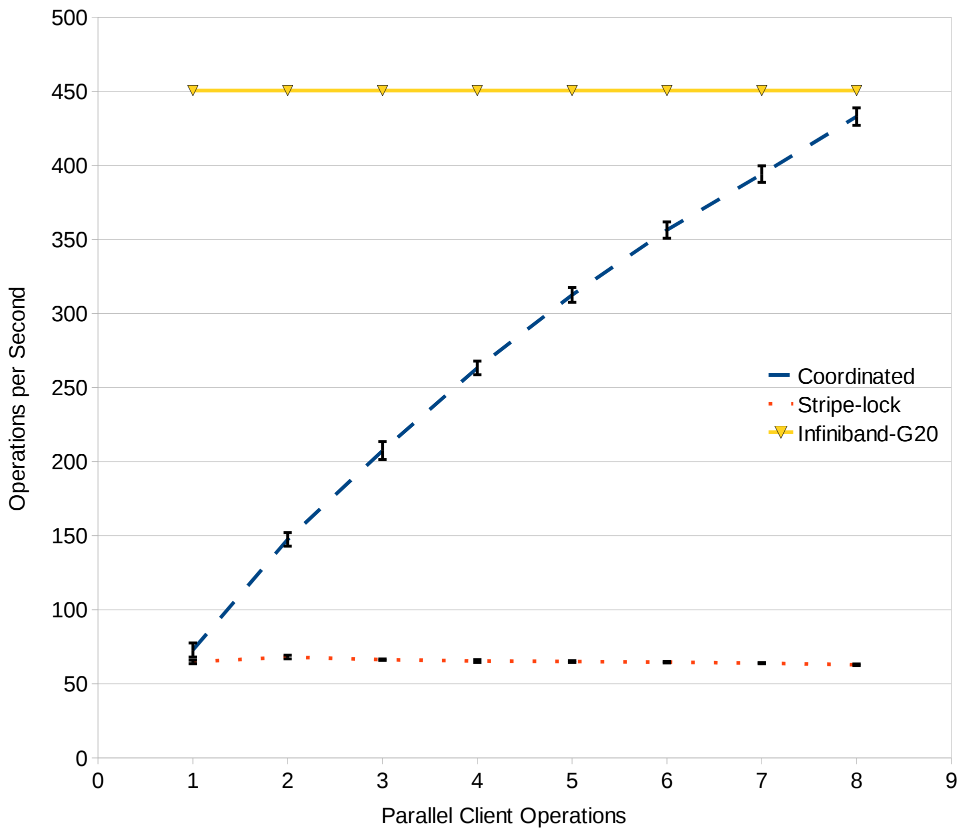

Fig. 32: stripe-lock mode vs. coordinated cluster mode over Infiniband-G20

7.200 RPM disk with a throughput of about 43 MB/sec evaluated using the disk I/O benchmark described in section 5.3.2. Evaluating the cluster on a system where the network interface outperforms the storage by a factor of 10 does not provide appropriate results. Therefore, I configured the data server to write into memory rather than to disk. In combination with the fast Infiniband-G20 network interface this benchmark should provide a good feeling for the potential of the coordinated cluster workflow.

The network throughput has been evaluation with the nuttcp tool and is about 450 MB/sec (ref. to table 26). Both the coordinated cluster mode and the stripe-lock mode have been tested with up to eight parallel operations of 1 MB each. Again the stripe-lock benchmark shows a constant throughput, whereas the coordinated cluster tends to saturate the network bandwidth. For eight parallel operations the coordinated cluster performs 432,987 ops/sec on average and therefore, saturates the network to 96,1 %. Furthermore, for a single client operation the coordinated cluster performs better than the stripe-lock mode. Looking back at the Suno results, the coordinated cluster writes also performed better there. In both configurations the network interface was the limiting factor.

	Ops/sec	
#-Clients	Stripe-lock mode	Coordinated cluster mode
1	64,786	72,824
2	68,064	147,521
3	66,316	207,423
4	65,415	263,286
5	65,072	312,560
6	64,623	356,399
7	63,944	394,107
8	62,889	432,987

Table 10: Benchmark results over Infiniband-G20

5.4.4. Luxembourg Results

The Granduc cluster at the site Luxembourg has a 10 Gbit Ethernet network, but only a single SAS hard drive.

Due to the bad performance of the disk I configured the necessary benchmarks to compare the two modes with the *tmpfs*. At Suno the storage and network performance of 150 MB/s and 112 MB/s are roughly balanced. At Granduc the 10 Gbit Ethernet has a bandwidth of about 425 MB/s (for details refer to table 33) and outperforms its storage by a factor of 11. It looks like the degree of hardware saturation is independent from the type of the device that becomes the bottleneck. Figure 33 shows the coordinated cluster and stripe-lock benchmark performance using *tmpfs* as storage backend.

It is not surprising that the performance is very good. Both, the coordinated cluster, and the stripe-lock mode perform about 75 operations per second at the single client benchmark. The further performance development for multiple client operations shows the same pattern already witnessed in figure 32. The stripe-lock benchmark's performance stays constant and the coordinated cluster's performance tends to saturate the hardware. These results confirm the ones retrieved by the tmpfs benchmark over the Infiniband network. The coordinated cluster run does saturate the 10 Gbit Ethernet to about 91,3%, whereas the Infiniband 20G run managed to achieve 96,1%. The next section is going to evaluate the latency of the single components and investigate the effects on the two operation modes.

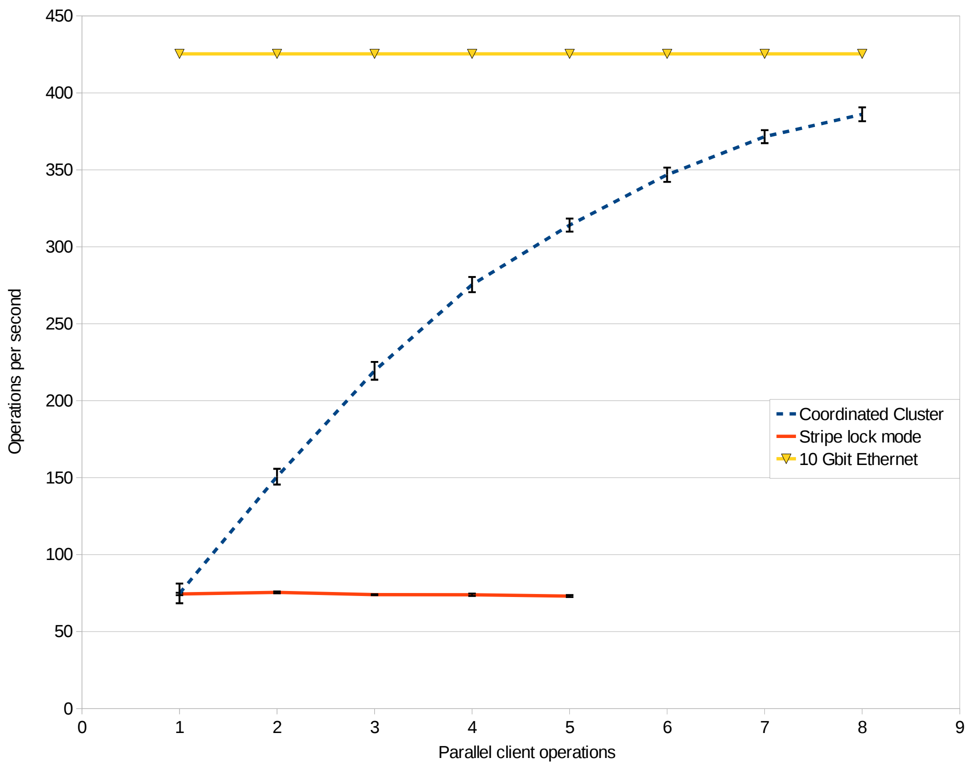

Fig. 33: Parallel write operations on ramdisk

5.5. Component Benchmarks At Suno

It's not surprising that the stripe-lock does not scale for conflicting stripe operations and therefore, is outperformed at highly parallel workloads. More surprisingly the performance of the coordinated cluster is even better for sequential operations at the *Suno* cluster. This means that the latency of a single operation of the coordinated mode is lower than the stripe-lock mode's latency. Figure 34 displays the measured latency of both modes along with its standard deviation.

Since the lock management does not provide a fair scheduling, the stripe-lock mode's deviation is increasing with the number of parallel operations. The latency of the stripe-lock mode increases linear with the number of parallel operations. This is caused by the missing parallelism. For a single client the average latency is about 32,9 ms, for two and three client operations 66,6 ms and 99,8 ms, respectively. For eight parallel operations the average latency reaches 275,9 ms. This is a total increase of 838,8%.

Looking at the latency of the coordinated cluster mode, we see that it is slightly influenced by the number of parallel operations. Table 9 shows the exact values. Starting with about 27,2 ms for a single client operation, it increases to 30,3 ms for two parallel operations and 33,0 ms for three parallel operations. This means an increase of 3,1 ms and 2,7 ms respectively. Three parallel operations perform about 87,5 ops/sec and saturate the parity server's

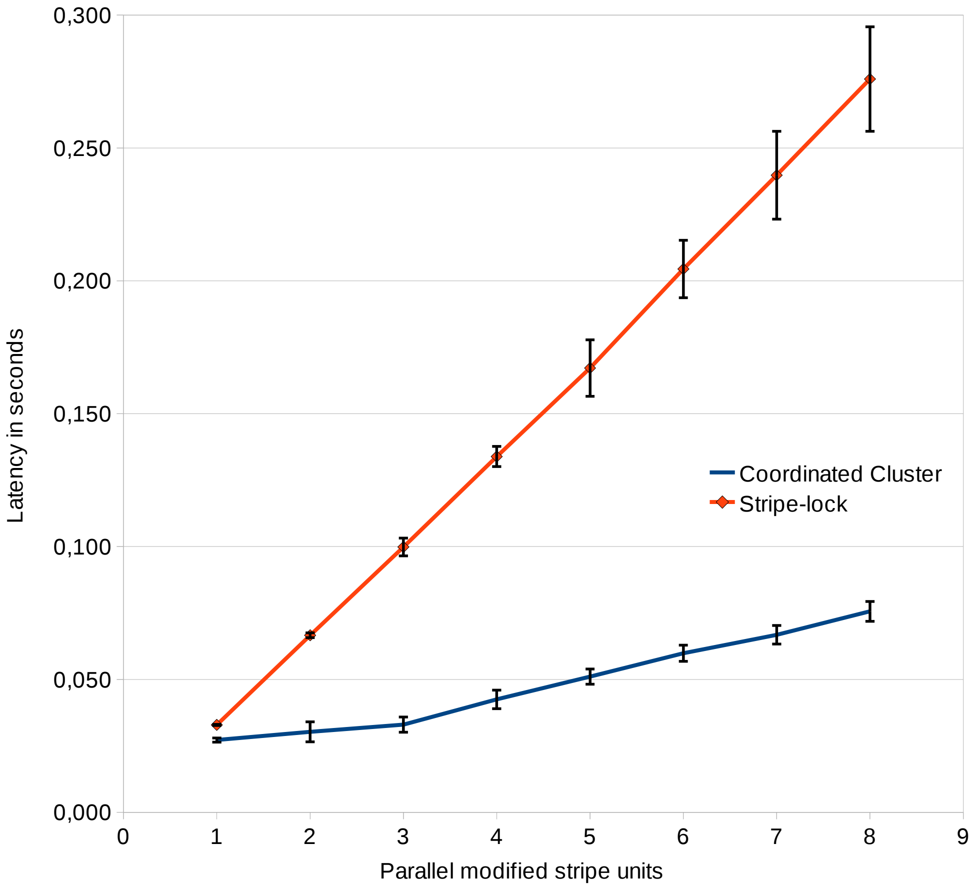

Fig. 34: Latency of the client operations at the Suno cluster

network connection to about 78,0 %. Adding now further clients working in parallel the latency increases by roughly 9 ms per client. This indicates that the network congestion has the highest impact on the latency. For eight parallel operations the average latency is 75,6 ms. This is about 278,1% with respect to a single client operation.

These latency results could be expected due to the throughput benchmarks. The next sections will investigate the effects of the individual actions of the two benchmark modes. Based on these results the single client operation modes are compared and the driving factors of the latency are identified.

5.5.1. Disk I/O Benchmark

The above presented results need to be explained. Investigating the individual steps of the workflows is a good way to do this. This section evaluates the single component benchmarks and the next section is going to compare the results of the single component benchmarks and the measured stripe-lock results. Table 11 represents the throughput and latency of the *suno* storage hardware.

#-Clients	MB/sec		Latency in ms	
	avg	σ	avg	σ in %
1	283,03	$\pm 9,8$	4,284	$\pm 40,3$
2	193,84	$\pm 23,0$	12,528	$\pm 25,4$
3	170,84	$\pm 10,6$	21,709	$\pm 8,6$
4	153,69	$\pm 10,0$	30,567	$\pm 10,3$
5	155,87	$\pm 8,8$	36,673	$\pm 3,0$
6	152,67	$\pm 6,2$	43,846	$\pm 2,9$
7	149,73	$\pm 4,7$	50,572	$\pm 2,9$
8	147,02	$\pm 4,4$	58,941	$\pm 1,7$

Table 11: Disk I/O bandwidth with parallel write operations

Back in figure 30 we already saw that the throughput of the storage is not the limiting factor. Since we are particularly interested in the latency of the single client operations, these latency results do provide further information. The data server write their data blocks in the same way, therefore, this benchmark gives a good estimate on the latency of the server's disk I/O. The single threaded benchmark measured an average latency of 4,284 ms. Adding further threads in parallel has a big impact on the latency. With every thread added the latency increases by about 8 ms to a total of 58,941 ms for eight parallel write operations. Parallel write operations on rotating disk devices usually decrease throughput. This effect gets even worse when the fsync syscall limits the capabilities of the disk controller to schedule the buffered write requests. The coordinated cluster benchmark has a latency of about 27,2 ms and therefore, the disks write latency of 4,284 ms is a considerable, but not the dominating factor.

5.5.2. CPU Parity Benchmark

Table 12 displays the result of the parity calculation benchmark performed on different systems.

CPU type	clock rate in MHz	time in sec	GB/sec	1 MB latency in ms	σ in %
Intel Xeon E5520	2.260	0,673	1.521,676	0,657	$\leq 0,1$
Intel Xeon L5420	2.493	0,605	1.691,206	0,591	$\leq 0,1$
Intel Xeon L5335	1.994	0,769	1.331,482	0,751	$\leq 0,1$
AMD Opteron 275	2.193	1,633	627,019	1,595	$\leq 0,1$

Table 12: Single core parity calculation

256 iterations of 4 MB each have been performed. A single core of a modern Intel Xeon processor can process about 1,5 GB/sec of XOR calculations. With today's common SMP architectures, the parity calculation is clearly not the limiting factor of this file system. The latency of processing a 1 MB block

is 0,657 ms at the E5520 at 2.260 MHz clock rate. However, the AMD does perform bad compared to the Intel processors. It has a front-side bus (FSB) rate of 400 MHz and only a 1024 KB cache. Compared to the 8 MB cache and 1066 MHz FSB of the E5520 CPU this explains the performance difference.

5.5.3. Data Server Message Ping Pong

Table 13 shows the time to send a message from one data server to another and process it. The full benchmark results of the 1 Gbit Ethernet (table 27), 10 Gbit Ethernet (table 28), the Myrinet-2000 (table 30) and Infiniband-G20 (table 29) runs can be found in the appendix.

Network Interface	Latency in ms	σ in %
1 Gbit Ethernet	0,080	$\pm 0,1$
10 Gbit Ethernet	0,073	$\pm 0,1$
Infiniband-G20	0,093	$\pm 0,2$
Myrinet-2000	0,080	$\pm 0,1$

Table 13: Data server messaging time via different network interfaces

The benchmark has been performed for the 1 Gbit and 10 Gbit Ethernet inferaces, the Myrinet-2000 interface and the Infiniband-G20. The latency is about 0,08 ms for the 1 Gbit Ethernet and the Myrinet-2000. Surprisingly, the Infiniband-G20 performs considerably worse than the 10 Gbit Ethernet interface. Back in section 5.4, at the coordinated cluster benchmark the Infiniband interface was saturated to about 96,1%, whereas the 10 Gbit Ethernet only achieved 91,3%. This indicates that the messaging during the coordinated cluster workflow has little impact on the overall write operation latency. The standard deviation has been calculated over 1.000 runs with 1.000 messages per run.

5.5.4. Stripe Unit Read Latency

The next table 14 displays the time to read a 1 MB data block from a data server's cache. A growing number of clients perform 1.000 read operations and calculate the average and standard deviation.

At the Suno cluster a single read operation has a latency of 10,633 ms and a throughput of about 92,764 MB/sec. This shows that a sequential read operation is unable to saturate the network bandwidth. Two parallel operations have a latency of 18,172 and a throughput of 106,014 MB/sec, which is about 94,5% of the 112,2375 MB/sec throughput measured with the nuttcp tool.

5.5.5. Lock Round-trip Time

Table 15 shows the results of the lock management round-trip benchmark. The measured latency represents the time spend by the clients during the stripe-lock mode workflow, to acquire and release the stripe-lock.

#-Clients	MB/s	Latency in ms	
		avg	σ in %
1	92,764	10,633	$\pm 0,2$
2	106,014	18,172	$\pm 8,0$
3	111,661	25,759	$\pm 15,3$
4	110,364	34,522	$\pm 10,6$
5	111,274	43,876	$\pm 8,0$
6	112,216	52,028	$\pm 8,6$
7	112,109	60,697	$\pm 9,1$
8	112,591	68,500	$\pm 11,8$

Table 14: Read operations from data server cache via 1 Gbit Ethernet

#-Clients	Latency in ms	
	avg	σ in %
1	0,416	$\pm 5,7$
2	1,270	$\pm 4,6$
3	2,017	$\pm 11,6$
4	2,889	$\pm 7,7$
5	3,505	$\pm 6,5$
6	4,065	$\pm 9,1$
7	5,440	$\pm 6,0$
8	5,300	$\pm 7,0$

Table 15: Lock acquire/release round-trip time at the Suno cluster

Especially if there are no conflicting requests, the latency of the acquire/release cycle is very low. Over the 1 Gbit Ethernet connection the round trip time is about 0,416 ms. Therefore, the impact on the stripe-lock mode benchmark is very low.

5.6. Stripe-lock Mode Analysis

All components of the stripe-lock workflow have been measured. These information are now compared with the measured results of the stripe-lock benchmark at the Suno cluster. Figure 35 shows these component's results. Clearly the data block transport over the network is the most expensive operation. So now we are going to investigate the effects on the stripe-lock workflow.

Based on the workflow in figure 27 I analyzed the steps performed and their degree of parallelism. Table 16 displays the single steps performed during the stripe-lock mode benchmark and sums up the latency of these operations as they were measured during the component benchmarks. First the client acquires the lock and then reads the old parity block. Then it calculates the new parity block. Here I assumed that the old data block was cached and is not read again. As a result the calculation of the operation delta is performed previously and

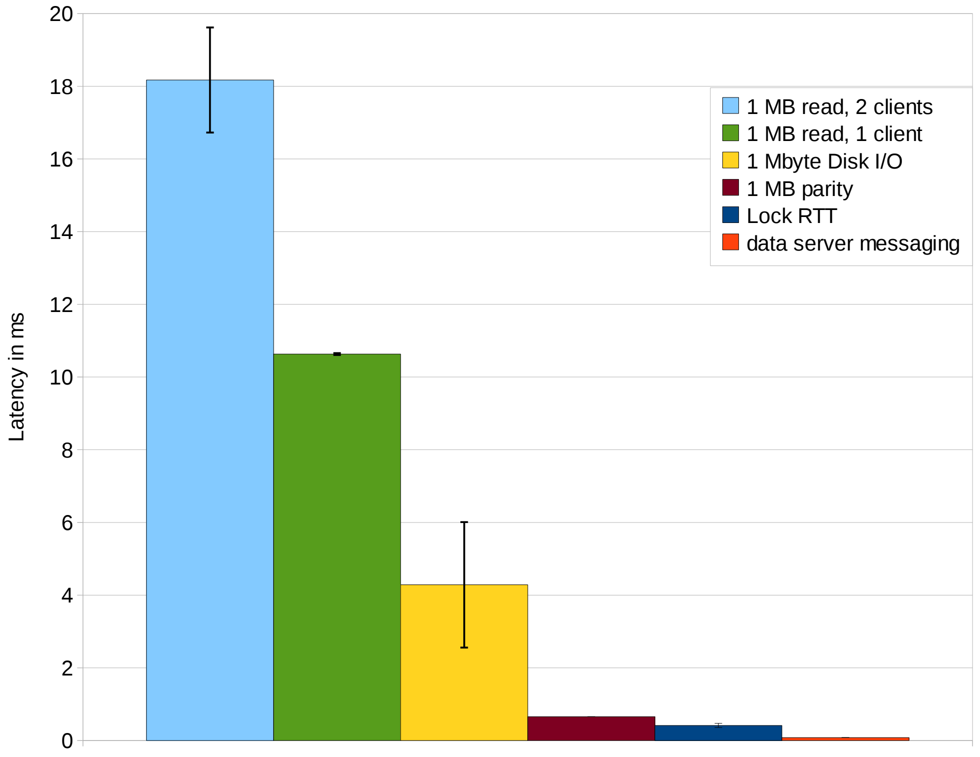

Fig. 35: Latency overview of the single component benchmarks

therefore, not part of the critical section. Otherwise, this would require a two message read operation of 18,172 ms latency instead of 10,633 and two parity calculations of 1,314 ms latency instead of 0,657. Then the data blocks are sent to the appropriate data server and these write the data to disk. After that, a response message is returned to the client. These three steps are performed by both data server independently and in parallel.

The measured latency of the stripe-lock mode for a single client operation is 32,897 ms. The difference of 1,345 ms is within the confidence interval of the single components. However, this confidence interval almost covers 10 % and is not very precise. The biggest deviation is most likely caused by the disk I/O operation, which has a variance of about 40 %. The single test results for the single threaded disk I/O benchmark are display in the appendix section 31. There we see that the average latency of the first 50 operations is 3,163 ms and the lowest latency measured is 2,870 ms. The stripe-lock benchmark writes 30,541 MB/sec, which is considerably lower than the disk's performance measured (ref. to table 11). The disks can be considered idle and therefore, the latency should match the lowest measured results. Setting the disk I/O value to 2,900 ms, the total latency adds up to 32,858.

In conclusion the stripe-lock workflow is dominated by the network performance since 28,805 ms or 84,1% of the 34,242 ms is spent by the data block transport. Of course the 1 Gbit Ethernet connection is not the first choice in

time in ms	Operation	Stripe-lock write operation in ms
0,416 $\pm 0,024$	lock acquire/release cycle	
10,633 $\pm 0,021$	read the old parity block	
0,657 $\pm 0,000$	calc new parity block	
18,172 $\pm 1,445$	send new data / parity block	
4,284 $\pm 1,726$	write to disk	
0,080 $\pm 0,000$	response message	
34,242 $\pm 3,216$	Sum	32,897 $\pm 0,197$

Table 16: Theoretical stripe-lock time

high performance computing. The 10 Gbit Ethernet interface installed at the Granduc cluster at the Site Luxembourg has a 5,467 ms latency (refer to table 34) for a single 1 MB block and 6,945 ms for two (refer to table 35). Combined this results in a latency of 12,412 ms and would clearly be a great performance improvement. However, still 69,5 % of the workflow's latency of then 17,849 ms would be caused by the data block transport.

5.7. Coordinated Cluster Mode Analysis

In the last section we saw the analysis of the stripe-lock mode and found out that the network traffic dominates the latency. Let's now look how this applies to the coordinated cluster mode.

Data server		Parity server	
time in ms	Operation	time in ms	Operation
10,633 $\pm 0,021$	Client sends data block		
0,080 $\pm 0,000$	received msg		
0,080 $\pm 0,000$	perform msg		
0.657 $\pm 0,000$	calc new parity block		
10,633 $\pm 0,021$	cancommit msg		
0,08 $\pm 0,000$	docommit msg		
4,284 $\pm 1,726$	write	0,657 $\pm 0,000$	parity calc
0,08 $\pm 0,000$	committed msg	4,284 $\pm 1,726$	write
4,944 $\pm 1,726$	max((4,284+0,08), (0,657+4,284))		
0,08 $\pm 0,000$	success msg		
0,08 $\pm 0,000$	success msg		
27,264 $\pm 1,768$	Sum		

Table 17: Theoretical coordinated cluster latency

First the client sends the new data block to the data server. Then the data server sends the received message to the coordinator and receives the preform response. On retrieval it calculates the delta of the operation and passes it to the coordinator. The coordinator starts phase two by sending the docommit message to the participant. Up to this point the operations were executed sequentially and the latency of the single tasks adds up. Now the parallel part starts. The participant writes the data to disk and sends the committed message to the coordinator. Meanwhile, the coordinator calculated the new parity object and writes it to disk. Finally, the coordinator replies the committed message with a success message and the participant forwards it to the client.

Based on the component latency results this workflow takes 27,264 ± 1,768 ms at the Suno cluster. 27,197 ms have been measured for the coordinated cluster single client benchmark, which is a precise match to the calculated one. However, it performs about 36,7 ops/sec and therefore, the disks must be considered idle too. Since I argued this way at the stripe-lock mode analysis, the disk I/O latency needs to be adapted to 2,900 ms too. Then the calculated latency is 25,880 ms, which is over a millisecond faster than the measured latency of the complete workflow, but still within the confidence interval.

I cannot fully explain the difference between the theoretical latency based on the single component benchmarks and the actually measured results. One reason I thought of are the garbage collector and time out monitoring threads. These threads monitor and modify the data structures used to organize the client operations. The threads block operations on a file while investigating the operations, but this only involves the traversal of a std::map. During the single client benchmark this map only contains the currently active operation and therefore, these threads can not influence the test results on a millisecond scale. Another explanation could be the delayed *fsync* at the parity server. The kernel buffer is flushed when the *committed* message of the participant arrived. Haven't the data been written to disk yet, it causes an additional delay and could explain the divergence.

Obviously the data block transport over the network is the dominating factor at the coordinated cluster workflow too. It takes 21,266 ms, which are about 78,0 % of the total latency of 27,264 ms. The big advantage of the coordinated cluster over the stripe-lock mode is the reduced number of data transports. After locking the stripe, the stripe-lock mode either reads the parity block or reads both the parity block and the old version of the data block the client is going to update. The new parity and data blocks are later sent to the appropriate data servers in order to be written to disk. This creates traffic in size of either three or four times the stripe unit size. In contrast to this the coordinated cluster always has exactly two times the stripe unit size traffic. This makes up for the increased coordination overhead.

We saw that the migration of responsibilities from the client side to the data server cluster reduced the network traffic and therefore, has positive impact on the operation's latency. Are there any downsides to this? Of course there are. First of all, the complete parity calculation has to be done by the data server cluster. The component benchmark showed that this is not a big deal for current SMP systems, nevertheless, in real life systems a lot of further tasks are performed by the data server that are not yet implemented. Currently the system does not provide error detection codes for the user data. Panasas

Tiered Parity Architecture[15] makes a good point on the upcoming challenges in providing end-to-end error correction. Calculating error correction codes will increase CPU load.

A further issue to mention here is the cache replacement algorithm, such as least recently used (LRU). This also creates additionally load to the CPU. I think that the CPU power of current multicore systems is sufficient to run the coordinated cluster. The cache size is a big issue too. A data server usually caches data in order to improve read performance and reduce disk accesses. In my architecture the data server calculate the difference of the new data block and the existing one and based on that create the new parity block. This implies that they have to read the existing data blocks and therefore, the cache is a crucial factor of the write process. Reading the existing data block from disk would not only add some milliseconds latency to the write process, but also creates a disk access that might conflict with other write operations.

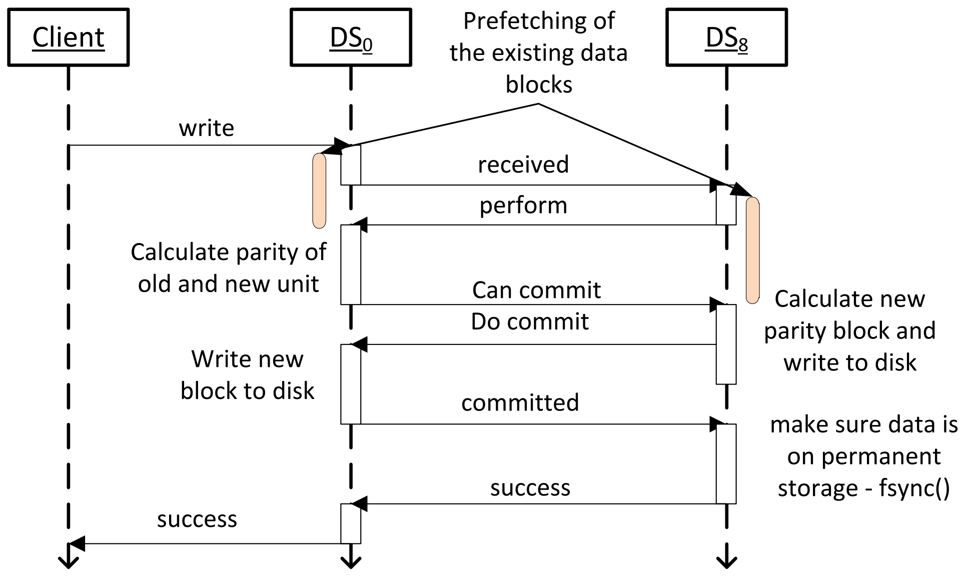
Client
DS_0
DS_8
Prefetching of the existing data blocks
write
received
perform
Calculate parity of old and new unit
Can commit
Do commit
Calculate new parity block and write to disk
Write new block to disk
committed
make sure data is on permanent storage - fsync()
success
success

Fig. 36: Prefetching potential of the coordinated cluster workflow

There are two techniques that might help to deal with this issue. First of all prefetching the existing data object could be used to hide the read latency. This can be integrated into the write workflow easily. Figure 36 shows the prefetching potential of the coordinated cluster workflow. DS_0 can prefetch the existing data block while waiting for the perform message. DS_8 can prefetch the existing parity block while waiting for the cancommit message of DS_0. Another possibility of improving the caching mechanism is the use of a SSD as a second cache layer. A SSD might already be used as a write buffer and could also act as a read buffer.

5.8. Full Stripe Write

The full stripe write of the coordinated cluster workflow, as described in section 5.2.4, has been evaluated at the Suno cluster. The group size is 8+1 and the stripe unit size is 1 MB. This means that the client has to calculate the parity block and send all data blocks plus the parity block to the data server cluster. During this evaluation we saw that the network traffic has the most influence on the performance. Here the bottleneck clearly is going to be the client's network interface.

avg ops/sec	avg latency in ms	σ in %
11,154	89,925	$\pm 7,419$

Table 18: Single client full stripe write performance via 1 Gbit Ethernet

Table 18 shows the result of a single client write operation. For each operation the client had to sent 9 MB of data to the cluster. At 11,154 ops/sec this creates a throughput of 100,386 MB/sec, which indicates that the network interface was the limiting factor.

Let's look at the latency of the single steps performed during the write operation displayed in table 19.

time in ms	Operation
5,256 ± 0,000	calculate parity block (8 × CPU parity)
80,187 ± 2,934	theoretical 9 MB transfer via 1 GbE
0,080 ± 0,000	cancommit message
0,080 ± 0,000	docommit message
4,284 ± 1,726	write to disk
0,080 ± 0,000	committed message
0,080 ± 0,000	success message
0,080 ± 0,000	success message
90,227 ± 4,660	

Table 19: Theoretical full stripe write latency

The confidence interval of the 9 MB data transfer latency has been taken from the 8 MB read operation performed at the Suno cluster. 90,227 ms per operation comes very close to the measured latency of 89,925 ms. On the one hand the latency of the I/O operation at idle disks must be considered lower than the averaged 4,284 ms. During the analysis of the stripe-lock mode I estimated the idle latency to be 2,900 ms, which results in a theoretical latency of 88,843 ms. On the other hand the 9 participating data server will not execute the single operation perfectly in parallel. This causes a higher latency of the single phases than the theory predicts and therefore, the measured latency of 89,925 ms is plausible.

5.9. Stripe Unit Size

Up to now the stripe unit size has always been 1 MB. This section evaluates the impact of different stripe unit sizes on the coordinated cluster performance. Figure 37 shows the impact of the stripe unit size on the data server throughput. The evaluation has been executed at the Suno cluster.

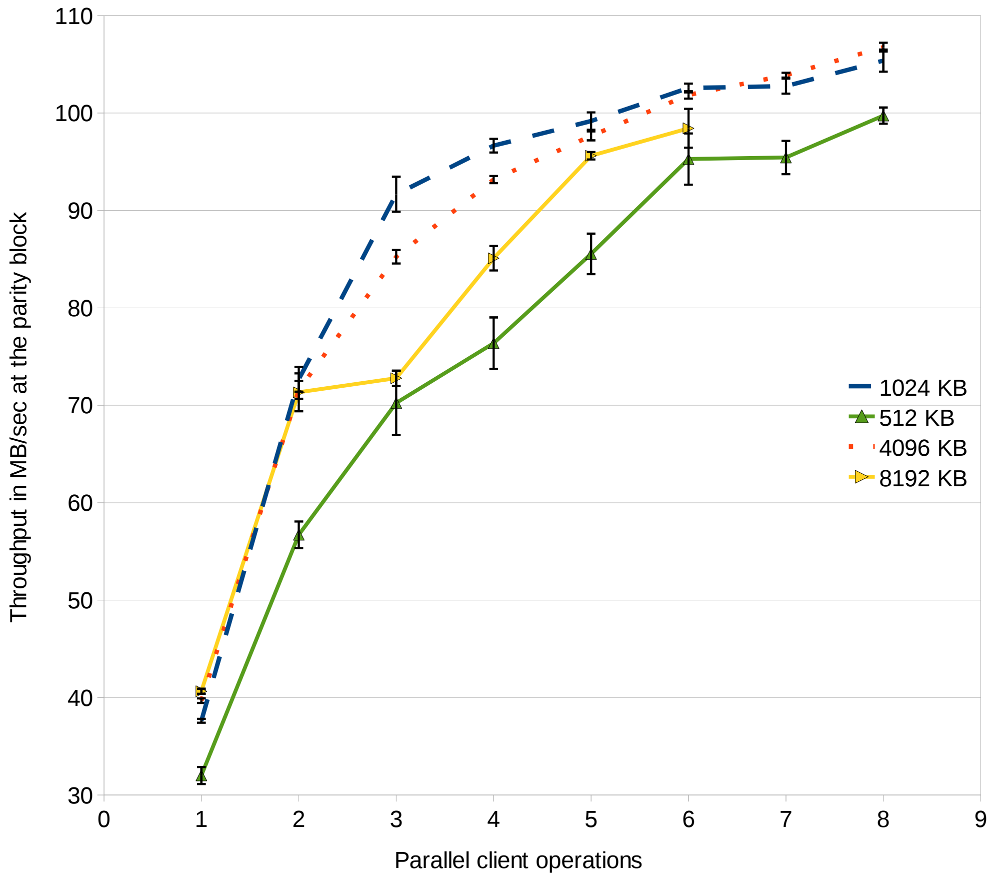

Fig. 37: Throughput on the parity block at different stripe unit sizes

The coordinated cluster workflow has been tested for up to eight clients and with stripe unit sizes of 512 KB, 1 MB, 4 MB and 8 MB. The throughput represents the data rate at the parity block. The graph shows that a unit size of 512 KB does not provide optimal performance. The large 8 MB size benchmark shows a dent at the three client value, which I can not explain. The 8 MB size run performed 500 operations and calculated the average throughput. Therefore, I do not think that this is caused by an measurement error. Nevertheless, a stripe unit size of 1-4 MB seems to provide good performance. Having the caching issue in mind, in general I would prefer the smaller unit size and have more smaller cache elements. In order to provide the best performance the cluster should be configured to fit the applications workflow best.

6. Future Work

The consistency model, the workflow and the architecture had been designed from scratch. The same holds for the data server and client library implementation. Therefore, it is not surprising that there are still a lot of issues to investigate.

6.1. Load Balancing

Maybe the most interesting part is the design of the load balancing mechanism. As mentioned in section 4.7, a lot of factors are to be considered. A basic load balancing starts at the metadata server with the distribution of new files. PVFS[6] is an example for a cluster file system that does not provide the migration of data between the data server. A more advanced loadbalancer should be able to perform data migration within the cluster and provide a good storage utilization and overall I/O performance. But it gets really interesting when we take the P-2-P architecture of the data server cluster into account. The data server can cooperate in order to compensate peak workloads on one of the data server. Write operations could be delegated to another data server, which then temporarily acts as the owner of the section. Another possibility would be a global cluster cache, which however, relies on a powerful cluster network.

6.2. Caching

In section 5.7 I already discusses the caching problem. Implementing an effective caching strategy requires a solid caching model. Usually a cache is used to provide better read performance, but in my architecture the cache is also involved in the write operation. Therefore, a simple LRU or LFU might not be the best choice. Furthermore, the introduction of a second cache layer, handled by a SSD, seems worth working on.

The effective utilization of the storage also needs further attention. A SSD drive or NVRAM could buffer incoming writes, which would be written back later in order to provide better disk throughput. This buffer could also be combined with techniques like parity logging[27]. Currently the data server writes a full data block every time, even though only a small portion of the block had been updated. Parity logging could reduce this overhead and a fast, non-volatile buffer could prevent frequently modified parity data from being written to disk every time.

6.3. Recovery Process

Currently the recovery process is not implemented. I described a possible workflow in section 4.5. Once a server failed, a second server crash would result in a data loss. On the one hand, the recovery time must be minimized in order to restore the needed redundancy. Therefore, the recovery process should try to utilize as much resources as possible. On the other hand, the client requests must not time out. From the applications point of view it makes no difference whether an operation failed due to a crashed server or it timed out because of the higher ranked recovery process. The load balancing mechanism has a great impact on the recovery process. In a sufficiently large file system, every data server is coordinating the recovery of some of the files of the failed server and therefore, a lot of individual loadbalancer instances have to be coordinated.

6.4. Snapshots

Snapshots and file versioning are features of a file system. A snapshot represents the current state of the file system, the moment the snapshot was taken. File versioning provides access to different versions of the same file. My consistency model using a global state management and a copy-on-write technique could provide these features. A snapshot could be taken without overhead and only requires the collection off all stripe's version vectors. Furthermore, the data objects, part of a snapshot, must not be garbage collected. The garbage collector would have to be modified in a way that an outdated data object is stored for a minimum amount of time, before being garbage collected. Otherwise, a data server could delete a local data object's version, which a moment ago, was marked by a snapshot at another data server.

6.5. End-to-End Integrity

According to Panasas[15] a full end-to-end integrity check is an upcoming demand in cluster computing. Therefore, a more sophisticated error detection and correction system might be needed in my architecture. It will be interesting to see, how this further load on the cluster interferes with the parity processing of the write operations.

Currently, the CPU load is not an issue in my architecture. However, here were several features discussed that might have a growing impact on the cluster's performance.

7. Conclusion

In this thesis I have shown that a coordinated data server cluster is able to guarantee file consistency while increasing the degree of parallelism. The evaluation in section 5.4 compared the write performance of the coordinated cluster with a locking approach managed by the metadata server. We saw that the write latency of a single client operation of both modes is about equal and even turns in favor of the coordinated mode if the network interface is the limiting factor. With an increasing number of conflicting client operations the stripe-lock prevents the parallel execution and no performance improvement can be seen for the metadata server managed workflow. In contrast to this the coordinated cluster is able to scale from 36,678 operations per second for one client, up to 105,088 ops/sec for eight parallel, conflicting client operations. Furthermore, the benchmarks showed that a RAID-4 configuration with eight stripe units and one parity block is able to saturate any network interface to over 90%. This clearly states that the workflow and consistency model scale with the hardware. This was achieved by trimming the critical section to a minimum. At the stripe-lock mode benchmark the critical section spans the complete time the lock was held, including disk I/O operations and network transfers. The coordinated workflow reduced the critical section to one parity calculation and a disk I/O operation.

Another advantage of the coordinated cluster is the cleaner allocation of responsibilities. The pNFS architecture requires the client library to monitor the state of its operation and either recover a failed operation or inform the metadata server about the error. In my architecture the client library is not responsible of the file's consistency. The metadata server, as described in the pNFS architecture, is not only part of the recovery process, but also of normal write operations. It is required to provide access control mechanisms to deal with the lost update problem on the parity block. Since the data server cluster performs this kind of access control internally, the metadata server is not involved in the data I/O processing. In consequence, in my architecture the metadata server is handling metadata operations, the data server cluster manages everything involved in the data processing and the client library has no responsibilities at all.

Interacting data server might be the next step in the evolution of object-based storage. The benchmark results showed that the system is able to match the performance of non-conflicting operations and considerably increase the throughput of conflicting operations. Furthermore, this architecture has a cleaner separation of concerns and provides a consistency guarantee. I think these are good reasons to advocate further research in this area.

Glossary

$\varnothing MQ$

$\varnothing MQ$[18] is a flexible, portable networking framework built by iMatrix[19]. It is specifically designed to handle very small messages efficiently and therefore is not the right choice for intensive I/O operations performed by a data server.

API

An API defines an interface that is used by interacting software components.

byte-range lock

A lock defined by an offset within the file and the length of the locked section. A client in posession of an BRL has exclusive write access on the specified range of a file.

capacity miss

The cache is full but a new cache entry needs to be inserted. An existing cache entry X is removed by some replacement strategy, such as First-Come-First-Serve. A subsequent request for X is called a capacity miss.

ClientSessionID

A globally unique ID assigned to a client during the mounting process. The CSID is represented by a 32 bit unsigned interger.

cold file

A cold file is a file that is accessed rarely.

copy-on-write

File systems using the copy-on-write mechanism do not overwrite an old data block with a new one. The new data is written to disk and the reference pointing to the old data block is replaced with the new data's reference. This provides a higher fault tolerance in case of incomplete write operations.

data object

A file data block with attached object's meta data.

data server

The data server offers read and write access to user data.

[18]http://www.zeromq.org
[19]http://www.imatix.com/

Einode

The Embedded Inode stores all metadata of a file. It consists of the Inode and the file system object's name

ext3

A journaled file system developed for the linux kernel and successor of ext2. Ext3 was added to the kernel version 2.4.15 in November 2001 and still is the default file system of the Debian distribution.

file object

An ordered set of stripe objects.

Filesystem in Userspace

FUSE allows the implementation of a fully functional file system in a user space program.

First-Come-First-Serve

Requests are handled in the order of their arrival.

hot-spare

A hot-spare server is a running data server that is not serving any files and waiting to take over the role of a failing server.

Inode

The metadata of a file system's object. It stores time stamps, the file layout, the objects mode, the link count, user and group id, the inode number and the files size.

InodeNumber

A globally unique 64 bit unsigned integer used to identify a file system object. The InodeNumber 1 is reserved for the root object '/'.

Metadata Server

A server specifically designed to handle distributed file system's metadata operations.

MTTR

The Mean Time To Repair defines the time spent until a failed device is replaced and the server continues it's service.

NFS

NFS is a protocol specifying a distributed file system. Multiple clients can access the remote file system via a computer network and operate on those files. The latest protocol version is NFS 4.1 (RFC5661)[22].

Object RAID

Object RAID is a term introduced by Panasas. Instead of grouping the servers into RAID groups, such as a RAID controller has a fixed group of disks, redundancy management is done on a per-file basis.

Object Storage Target

In Lustre an Object Storage Target (OST) is an interface to a single exported backend volume. It is conceptually similar to an NFS export, except that an OST does not contain a whole namespace, but rather file system objects[20].

OperationID

A unique ID for a client operation.

pNFS

The parallel Network File System is an optional part of the latest version of the NFS protocol (RFC5661)[22]. It splits the handling of file system operations, such as create, read or delete, from the data I/O handling. A specifically designed MDS is performing the file system operations and multiple DS perform data I/O in parallel. This increases both the data server global capacity and the I/O throughput for a single operation. Furthermore, the metadata server can perform operation on large directories with millions of file more efficiently than a single server NFS that is also performing I/O operations.

POSIX

The Portable Operating System Interface (POSIX) is a family of standards specified by the IEEE. These standards are designed for maintaining compatibility between operating systems.

RAID-0

In a RAID-0 all disks write different chunks without creating redundant or replicated data. This offers a great performance improvement, but also increases the risk of data loss.

RAID-1

A RAID-1 consists of two disk storing the same data.

RAID-4

In a RAID-4 environment of n disks, n-1 disks store user data and one disk stores the parity data. Since every write operation creates an operation on the parity data, this algorithm creates a high load on the parity disk.

[20] http://wiki.lustre.org/index.php/FAQ_-_Fundamentals

Redundant Array of Inexpensive Disks

Several independent disk drives are treated as a virtual disk. The disks are either connected to a specific RAID controller (hardware RAID) or the operating system combines the available disks (software RAID). In a RAID system the data of a file is split into chunks and these are scattered to the disks.

Remote Procedure Call

A RPC is a mechanism to execute subroutines within another address space. The client sends the necessary parameter to the server offering a particular service. The server performs the routine and returns the result to the client.

Round-trip time

The time spent between a request message and the return of the response message.

SMP

A symmetric multiprocessor is an architecture where two or more identical processors work in parallel and share the system's hardware resources, such as main memory.

Solid State Disk

A SSD is a hard drive without moving objects, such as rotating disks. Especially for random accesses SSDs provide a shorter access latency.

stripe object

A collection of DO with an attached parity object.

stripe-lock

A lock on the parity object of a stripe, which is thereby granting exclusive write access to the whole stripe.

three-phase commit protocol

The three-phase commit protocol[25] was designed to provide a non-blocking workflow and overcome issues of the two-phase commit protocol.

tmpfs

A temporary file system that writes into main memory and is mounted on /dev/shm.

two-phase commit protocol

The two-phase commit protocol[12] is a type of atomic commitment protocol. It involves a coordinator and several participants. In phase one the coordinator asks the participants whether they are able to perform the operation. Depending on the result he coordinator sends a doCommit message to perform the operation or doAbort in order to abort the operation.

Acronyms

$\oslash MQ$	Zero Message Queue
2PC	two-phase commit protocol
3PC	three-phase commit protocol
BRL	byte-range lock
COW	copy-on-write
CRC	cyclic redundancy check
CSID	ClientSessionID
DO	data object
DS	data server
FO	file object
FSB	front-side bus
FUSE	Filesystem in Userspace
GB	gigabyte
IETF	Internet Engineering Task Force
KB	kilobyte
LRU	least recently used
MB	megabyte
MDS	metadata server
MTTR	mean time to repair
NFS	Network File System
OpID	OperationID
OST	Object Storage Target
pNFS	parallel Network File System
RAID	Redundant Array of Inexpensive Disks
RPC	remote procedure call
RTT	round-trip time

SMP	symmetric multiprocessing
SO	stripe object
SSD	solid state disk

References

[1] Symantec Survey Reveals Digital Information Costs Businesses $1.1Trillion. http://www.symantec.com/about/news/release/article.jsp?prid=20120625_01.

[2] Khalil Amiri, Garth A. Gibson, and Richard A. Golding. Highly concurrent shared storage. In *ICDCS*, pages 298–307, 2000.

[3] J. Zelenka B. Halevy, B. Welch. Object-based parallel nfs (pnfs) operations. Standards Track RFC 5664, Internet Engineering Task Force (IETF), 2010. http://tools.ietf.org/html/rfc5664.

[4] D. Black, S. Fridella, EMC Corporation, J. Glasgow, and Google. Parallel nfs (pnfs) block/volume layout. Standards Track RFC 5663, Internet Engineering Task Force (IETF), 2010. http://tools.ietf.org/html/rfc5663.

[5] Pei Cao, Swee Boon Lim, Shivakumar Venkataraman, and John Wilkes. The tickertaip parallel raid architecture. *ACM Transactions on Computer Systems (TOCS*, 1994.

[6] Philip H. Carns, Walter B. Ligon, III, Robert B. Ross, and Rajeev Thakur. Pvfs: A parallel file system for linux clusters. In *In Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327. USENIX Association.

[7] M. Eisler. Metadata striping for pnfs. Internet-draft, 2010. http://tools.ietf.org/html/draft-eisler-nfsv4-pnfs-metastripe-02.

[8] Tais B. Ferreira, Rivalino Matias, Autran Macêdo, and Lucio B. Araujo. A comparison of memory allocators for multicore and multithread applications: A quantitative approach. In *Proceedings of the 2011 Brazilian Symposium on Computing System Engineering*, SBESC '11, pages 200–205, Washington, DC, USA, 2011. IEEE Computer Society.

[9] Daniel Ford, Francois Labelle, Florentina Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. Availability in globally distributed storage systems. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[10] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[11] Garth Gibson. Advances in raid and hpc storage reliability. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[12] J. Gray. Notes on data base operating systems. In R. Bayer, R. Graham, and G. Seegmüller, editors, *Operating Systems*, volume 60 of *Lecture Notes in Computer Science*, pages 393–481. Springer Berlin / Heidelberg, 1978. 10.1007/3-540-08755-9_9.

[13] Jim Gray and Leslie Lamport. Consensus on transaction commit. *ACM Trans. Database Syst.*, 31(1):133–160, March 2006.

[14] Emmanuel Jeanvoine and David Margery. Introduction to grid'5000, 2011. http://www.grid5000.fr/mediawiki/images/Seminaire_intro.pdf.

[15] Larry Jones, Matt Reid, Marc Unangst, Garth Gibson, and Brent Welch. Panasas tiered parity architecture, 2010.

[16] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[17] Larry Lancaster and Alan Rowe. Measuring real-world data availability. In *Proceedings of the 15th USENIX conference on System administration*, LISA '01, pages 93–100, Berkeley, CA, USA, 2001. USENIX Association.

[18] Frederic P. Miller, Agnes F. Vandome, and John McBrewster. *Cyclic Redundancy Check: Computation of CRC, Mathematics of CRC, Error detection and correction, Cyclic code, List of hash functions, Parity bit, Information ... Cksum, Adler- 32, Fletcher's checksum.* Alpha Press, 2009.

[19] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[20] Version 2 documentation Parallel Virtual File System. http://www.pvfs.org/cvs/pvfs-2-7-branch.build/doc/pvfs2-guide/pvfs2-guide.php#SECTION00036000000000000000.

[21] David A. Patterson, Garth A. Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (raid). In *SIGMOD Conference*, pages 109–116, 1988.

[22] D. Noveck S. Shepler, M. Eisler. Network file system (nfs) version 4 minor version 1 protocol. Standards Track RFC 5661, Internet Engineering Task Force (IETF), 2010. http://tools.ietf.org/html/rfc5661.

[23] Ed. S. Shepler, Inc Storspeed, Ed M. Eisler, Ed D. Noveck, and NetApp. Network file system (nfs) version 4 minor version 1 external data representation standard (xdr) description. Standards Track RFC 5662, Internet Engineering Task Force (IETF), 2010. http://tools.ietf.org/html/rfc5662.

[24] Frank Schmuck and Roger Haskin. Gpfs: A shared-disk file system for large computing clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST '02, Berkeley, CA, USA, 2002. USENIX Association.

[25] D. Skeen and M. Stonebraker. A formal model of crash recovery in a distributed system. *IEEE Trans. Softw. Eng.*, 9(3):219–228, May 1983.

[26] Daniel Stodolsky, Garth Gibson, and Mark Holland. Parity logging overcoming the small write problem in redundant disk arrays. *SIGARCH Comput. Archit. News*, 21(2):64–75, May 1993.

[27] Daniel Stodolsky, Mark Holland, William V. Courtright, II, and Garth A. Gibson. Parity logging disk arrays. *ACM Trans. Comput. Syst.*, 12(3):206–235, August 1994.

[28] Lustre Architecture Server Network Striping. `http://wiki.lustre.org/index.php/Architecture_-_Server_Network_Striping#SNS_Fundamentals`.

[29] Peter Ungaro. Tackling the next generation of hpc challenges..., 2012.

[30] Kashi Venkatesh Vishwanath and Nachiappan Nagappan. Characterizing cloud computing hardware reliability. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 193–204, New York, NY, USA, 2010. ACM.

[31] Sage A. Weil. *Ceph: reliable, scalable, and high-performance distributed storage.* PhD thesis, Santa Cruz, CA, USA, 2007. AAI3288383.

[32] Sage A. Weil. Ceph: distributed storage for cloud infrastructure, 2012.

[33] Brent Welch, Marc Unangst, Zainul Abbasi, Garth Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. Scalable performance of the panasas parallel file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 2:1–2:17, Berkeley, CA, USA, 2008. USENIX Association.

A. Appendix

1GB	time	GB/s
1	0,675373	1516,2
2	0,672863	1521,86
3	0,672834	1521,92
4	0,672884	1521,81
5	0,672847	1521,89
6	0,672781	1522,04
7	0,672755	1522,1
8	0,672768	1522,07
9	0,672744	1522,12
10	0,672787	1522,03
11	0,672775	1522,05
12	0,672794	1522,01
13	0,672756	1522,1
14	0,6728	1522
15	0,672812	1521,97
16	0,672766	1522,07
17	0,672808	1521,98
18	0,672856	1521,87
19	0,672906	1521,76
avg	0,6729425789	1521,6763157895

Table 21: Parity calculation on one Intel Xeon E5520 core at 2.26 GHz

1 Gbit Ethernet (Broadcom NetXtremeII BCM5716) nuttcp results.
cmd: nuttcp -i1 -w8m 138.96.21.142
112.1250 MB / 1.00 sec = 940.5228 Mbps 0 retrans
112.2500 MB / 1.00 sec = 941.6165 Mbps 0 retrans
112.1875 MB / 1.00 sec = 941.1111 Mbps 0 retrans
112.1875 MB / 1.00 sec = 941.0273 Mbps 0 retrans
112.1875 MB / 1.00 sec = 941.1148 Mbps 0 retrans
112.2500 MB / 1.00 sec = 941.6815 Mbps 0 retrans
112.1875 MB / 1.00 sec = 941.0678 Mbps 0 retrans
112.1875 MB / 1.00 sec = 941.0960 Mbps 0 retrans
112.2500 MB / 1.00 sec = 941.5892 Mbps 0 retrans
112.1875 MB / 1.00 sec = 941.1591 Mbps 0 retrans
1122.3750 MB / 10.00 sec = 941.1475 Mbps 5 %TX 16 %RX 0 retrans 0.10 msRTT height

Table 22: Nuttcp results on 1 Gbit Ethernet

Myrinet-2000 (M3F-PCIXF-2)
nuttcp -i1 -w8m 192.168.7.80
235.8125 MB / 1.00 sec = 1978.0417 Mbps 0 retrans
235.7500 MB / 1.00 sec = 1977.6618 Mbps 0 retrans
235.8125 MB / 1.00 sec = 1978.1426 Mbps 0 retrans
235.8125 MB / 1.00 sec = 1978.1208 Mbps 0 retrans
235.8125 MB / 1.00 sec = 1978.1525 Mbps 0 retrans
235.8125 MB / 1.00 sec = 1978.1406 Mbps 0 retrans
235.8125 MB / 1.00 sec = 1978.1090 Mbps 0 retrans
235.8125 MB / 1.00 sec = 1978.1485 Mbps 0 retrans
235.8125 MB / 1.00 sec = 1978.1386 Mbps 0 retrans
235.7500 MB / 1.00 sec = 1977.6400 Mbps 0 retrans
2358.2500 MB / 10.00 sec = 1978.0306 Mbps 14 %TX 25 %RX 0 retrans 0.04 msRTT

Table 23: Nuttcp results on Myrinet-2000

#-parallel threads	MB/sec	Latency in ms
1	26,5656	0,1472708899
2	33,34352	0,6852361688
3	36,10524	0,384954196
4	39,50472	0,387234757
5	39,65208	1,9220508986
6	40,7332	1,3890554651
7	40,89312	1,4669753294

Table 24: Helios Disk I/O benchmark resuls, 2x73 GB RAID-0

	Coordinate Cluster		Stripe-lock	
#-Clients	Ops/s	σ	Ops/s	σ
1	11,836	$\pm 0,118$	21,727	$\pm 0,062$
2	21,728	$\pm 0,140$	20,657	$\pm 0,093$
3	28,278	$\pm 0,159$	18,623	$\pm 0,086$
4	32,811	$\pm 0,388$	17,803	$\pm 0,064$
5	32,385	$\pm 0,498$	19,167	$\pm 0,091$
6	34,612	$\pm 0,351$	18,547	$\pm 0,075$
7	36,634	$\pm 0,247$	18,206	$\pm 0,285$
8	36,307	$\pm 0,306$	17,316	$\pm 0,208$

Table 25: Coordinated cluster amd stripe-lock results at the Helios HDD

```
Inniband-G20 (MHGH29-XTC)
nuttcp -i1 -w8m 172.18.65.82
451.5000 MB / 1.00 sec = 3787.0892 Mbps 0 retrans
450.3125 MB / 1.00 sec = 3777.7670 Mbps 0 retrans
451.2500 MB / 1.00 sec = 3785.3631 Mbps 0 retrans
450.0625 MB / 1.00 sec = 3775.4281 Mbps 0 retrans
450.5625 MB / 1.00 sec = 3779.4410 Mbps 0 retrans
450.3125 MB / 1.00 sec = 3777.6121 Mbps 0 retrans
451.3750 MB / 1.00 sec = 3786.3284 Mbps 0 retrans
450.1875 MB / 1.00 sec = 3776.5635 Mbps 0 retrans
450.2500 MB / 1.00 sec = 3776.9783 Mbps 0 retrans
449.5625 MB / 1.00 sec = 3771.1696 Mbps 0 retrans
4505.3750 MB / 10.00 sec = 3779.1395 Mbps 51 %TX 44 %RX 0 retrans 0.09 msRTT
```

Table 26: Nuttcp results on Griffon Infiniband-G20 network

Latency in ms		
min	avg	max
0,075945	0,0822232	0,179265
0,075433	0,0796788	0,094052
0,075452	0,0799514	0,093857
0,075426	0,0787228	0,084337
0,076018	0,0804117	0,088221
0,07444	0,0807508	0,09464
0,075505	0,0788086	0,094893
0,074256	0,0796521	0,090319
0,075138	0,0784063	0,085063
0,07559	0,0789408	0,089376
0,075792	0,080574	0,093491
0,076823	0,0816384	0,094437
0,077243	0,0820279	0,096101
0,076015	0,0807839	0,096209
0,076036	0,0810916	0,094463
0,076532	0,0821212	0,098996
0,07612	0,0811726	0,100444
0,076414	0,0803002	0,095347
0,07537	0,0803173	0,098571
0,075276	0,0804536	0,100153
0,0757412	0,08040136	0,09811175

Table 27: Data server messaging time over 1 Gbit Ethernet

Latency in ms		
min	avg	max
0,070545	0,0752979	0,196713
0,071026	0,071791	0,073236
0,070486	0,0723294	0,074244
0,069987	0,0716084	0,074816
0,073232	0,0740934	0,075754
0,070495	0,0724682	0,074867
0,070196	0,070934	0,07259
0,070198	0,0712169	0,07319
0,070437	0,0715312	0,074651
0,070289	0,0713346	0,074113
0,0706891	0,0722605	0,0864174

Table 28: Data server messaging time over 10 Gbit Ethernet

Latency in ms		
min	avg	max
0,089803	0,097663	0,180015
0,08987	0,0947665	0,098119
0,0894	0,0945483	0,098432
0,089914	0,093838	0,096475
0,088103	0,0946474	0,099598
0,087325	0,0933553	0,098641
0,088656	0,0941754	0,09859
0,086298	0,0918142	0,09667
0,086114	0,0915714	0,09755
0,086306	0,0917864	0,09598
0,087168	0,0913049	0,095067
0,085873	0,0905732	0,095489
0,088949	0,093272	0,09669
0,087203	0,0929604	0,098161
0,089178	0,0922377	0,096415
0,087055	0,0921545	0,095998
0,088663	0,092157	0,096199
0,088678	0,0924359	0,09629
0,088685	0,0925324	0,096665
0,08925	0,0927197	0,096764
0,08812455	0,09302568	0,1011904

Table 29: Data server messaging time over Infiniband-G20

Latency in ms		
min	avg	max
0,078562	0,082237	0,141849
0,078572	0,0794262	0,08046
0,078568	0,0793681	0,080137
0,077991	0,0793209	0,080194
0,078209	0,0794251	0,081055
0,077821	0,0792111	0,080078
0,078246	0,0793315	0,080456
0,078309	0,0794724	0,080503
0,078143	0,079581	0,088781
0,078143	0,0791719	0,080562
0,077925	0,0791739	0,080244
0,078364	0,079354	0,080664
0,078281	0,0791332	0,079786
0,078396	0,0794038	0,08032
0,078282	0,0794061	0,084241
0,078177	0,0794639	0,080534
0,0785	0,0792826	0,080214
0,078359	0,0795139	0,088218
0,078176	0,0796068	0,080599
0,077444	0,0792048	0,080537
0,0782234	0,07950441	0,0844716

Table 30: Data server messaging time over Myrinet-2000

			Latency in ms		
Ops	secs	MB/s	min	avg	max
50	0,158284	315,8878977029	0,00287	0,00316296	0,004233
100	0,328993	303,9578349691	0,002847	0,00341044	0,024279
150	0,490921	305,5481431839	0,003014	0,00323528	0,003587
200	0,672405	297,4397870331	0,003167	0,0036264	0,010799
250	1,01574	246,1259771201	0,003186	0,0068633	0,024052
		283,0379691124	0,0031223333	0,0045749933	0,0128126667

Table 31: Single client 1 MB write operations

			Latency in ms		
Ops	secs	MB/s	min	avg	max
50,	0,561486	356,198	0,003516	0,0112748	0,075232
100	2,17858	183,606	0,003715	0,0323146	0,578635
150	3,70366	162,002	0.00345	0,0305215	0,281982
200	5,35839	149,299	0,004667	0,0332519	0,406831
250	6,67682	149,772	0,003537	0,026181	0,411
		161,169	0,0031223333	0,0045749933	0,0128126667

Table 32: Four parallel client 1 MB write operations

cmd: nuttcp -i8 -w1m 192.168.14.12
452.8125 MB / 1.00 sec = 3798.0146 Mbps 0 retrans
420.5000 MB / 1.00 sec = 3527.1839 Mbps 0 retrans
421.9375 MB / 1.00 sec = 3539.7054 Mbps 0 retrans
421.2500 MB / 1.00 sec = 3533.4538 Mbps 0 retrans
419.8750 MB / 1.00 sec = 3522.7621 Mbps 0 retrans
428.9375 MB / 1.00 sec = 3598.1526 Mbps 0 retrans
422.1250 MB / 1.00 sec = 3540.7012 Mbps 0 retrans
420.4375 MB / 1.00 sec = 3526.6244 Mbps 0 retrans
424.1875 MB / 1.00 sec = 3558.3177 Mbps 0 retrans
420.8750 MB / 1.00 sec = 3531.2228 Mbps 0 retrans
4253.3750 MB / 10.00 sec = 3567.4573 Mbps 26 %TX 32 %RX 0 retrans 0.06 msRTT

Table 33: Nuttcp results on 10 Gbit Ethernet

			Latency in s		
Ops	secs	MB/s	min	avg	max
50	0,319289	156,598	0,004053	0,00544306	0,008185
100	0,593215	168,573	0,00463	0,00547692	0,005994
150	0,866505	173,109	0,004719	0,00546472	0,005894
200	1,13915	175,569	0,004735	0,00545178	0,006204
250	1,41526	176,646	0,004875	0,00552082	0,006106
300	1,69038	177,475	0,004739	0,0055015	0,005862
350	1,96664	177,969	0,004823	0,00552406	0,005864
400	2,23926	178,631	0,004832	0,00545134	0,005999
450	2,512	179,14	0,004761	0,00545372	0,007104
500	2,78132	179,771	0,00472	0,00538534	0,005963
			0,0046887	0,005467326	0,0063175

Table 34: Singel client read operation over 10 Gbit Ethernet

			Latency in s		
Ops	secs	MB/s	min	avg	max
50	0,493757	202,529	0,004017	0,0113168	0,413661
100	0,833187	240,042	0,004676	0,0064576	0,009183
150	1,18493	253,179	0,005407	0,00662708	0,00912
200	1,52916	261,581	0,004816	0,00658311	0,008364
250	1,87563	266,578	0,005153	0,00672271	0,008438
300	2,2085	271,678	0,004982	0,00645665	0,009569
350	2,54606	274,934	0,005049	0,00660779	0,007949
400	2,87145	278,604	0,004991	0,00642902	0,007838
450	3,18618	282,47	0,004772	0,00610461	0,009521
500	3,50133	285,605	0,004565	0,00614624	0,010059
			0,0048428	0,006945161	0,0493702

Table 35: Double client read operation over 10 Gbit Ethernet