Data Consistent Up- and Downstreaming in a Distributed Storage System

Peter Sobe
University of Lübeck, Germany
Institute of Computer Engineering
e-mail: sobe@iti.uni-luebeck.de

***Abstract*— Distribution of large data objects among several storage servers is a common technique to speed up access rates. In combination with parity schemes, failures of single server nodes can be tolerated, so that such systems reach a certain degree of fault tolerance. In this paper such a distributed server system is analyzed. Data objects are stored in a data layout according to RAID level 3 among disk subsystems of different computers. An access control provides concurrent up- and down-streaming of data objects to/from the distributed storage system with ensured data consistency. This consistency control is described in combination with the handling of faulty server nodes and faulty clients. Furthermore, performance is measured with several access patterns. An application of that technique is for instance a distributed video server, allowing permanently updates without interrupting access.**

I. Introduction

Nowadays, storage and streaming servers for digital media are substituting video tape archives and classical distribution methods of media. By storing media files on disks, durable media storages can be built by exploiting fault tolerant data layout schemes. Additionally, such storage servers scale with the growing amount of content and access load. Such server systems for storage and delivery of media content are now in the focus of research.

In order to store data in a reliable way and to permit high data access rates, it is obvious to apply RAID[12]-like techniques. But exclusively using a single host-based disk array (RAID) does not provide scalability within distributed server systems. The point is that a disk array is accessed over one single active controller which quickly becomes a bottleneck in terms of availability and performance. Thus, current scalable storage systems do either employ a separate network for global access to storage devices, which itself usually are RAID systems, or use the communication network of clustered computers to combine many disks of several nodes to a scalable and fault tolerant storage system. The first class is known as Storage Area Networks (SAN e.g. [11]). The second class, which is in the focus of this study, are distributed software systems that employ RAID-like data layout schemes over several hosts in a network. Multiple disks are logically coupled over an existing network and data is striped over disks of the coupled computers. This data distribution requires transfer of data to and from several hosts. The network in the path between storage device and application does not slow down the access rate to a single storage medium, as long as the data rate over the network is not lower than the storage access rate and secondly, disk accesses and data transfer over the network interface are overlapped. It is feasible to reach faster access by using multiple disk subsystems and file systems in parallel which also provides tolerance against single disk and node failures. Examples for systems applying such a technique are given in [1], [5], [7], [8], [14]. For this study, we used an implementation of a RAID-level-3-like storage scheme which was developed by ourselves, called *netRAID*. It has already been conceptually introduced in [16]. In this paper, the ability to deal with faulty clients is integrated into the access control algorithm. Furthermore, performance results of the system are reported, in order to prove the concept and to analyze the extra cost of the access control.

The need for consistency can be described roughly in the following. Ordinary host-attached RAID systems offer a single entry point to the disk array. This point must be passed by all read and write operations that cause accesses to subblocks on the disks. In that way, a single disk controller determines the order of operations and data consistency is kept in a way we would expect from a single disk. Accordingly, accesses are driven by a single local file system. In a distributed implementation as here, several accessing processes may read and alter striped data objects on different hosts, so that consistency is not guaranteed. Without controlling access operations, read data may be corrupted. In case of concurrent updates, data may also kept stored in a corrupted state. Commonly, this problem is solved by locking entire data objects during updates. Many systems use a meta-server to store information and to manage locking of data in case of writing access. Another solution is to use a log–based file system, where altered data is entirely rewritten to new data objects, such as [1]. We suggest a technique that allows concurrent updates without locking data objects and without any need to store a new version of the entire data object.

The described solution does not significantly cost performance. But it induces some side effects that on the one hand are valuable for storing data with sequential access. On the other hand, we noticed a vulnerability against failures of accessing processes that fortunately can be de-fused by relatively simple extensions.

Related work and other storage solutions are described in section II. The application scenario is described in section III and the fault model in IV, whereas information about the used cluster system and software environment can be found in V.

Beside the distribution scheme, we give a detailed description of our solution to guarantee consistency in section VI and analyze how faulty server or client nodes are tolerated. The performance of the system is reported in section VII under several access patterns.

II. Related Work

There can be found a lot of systems employing distributed computer systems with local disk storage for a coupled storage system. Some of them were developed in a context of video storage and delivery platforms.

Storage strategies for video servers were discussed in [9]. Two proposed data layout schemes take the MPEG-1 frame coding into account. In one proposed variant, non reference frames are stored in round robin manner among the cluster nodes without parity information, because their lost only causes a limited quality degradation. This decreases the storage overhead by 30%. A second proposed variant suggests alternative B-frames instead of parity information for all reference frames. In that way, no parity information is used at all. It requires a modification of the video encoder to generate alternative frames but can save disk capacity against a non coding-aware RAID-like parity scheme. This adapted storage scheme would be compatible with the suggested local ordered access scheme.

RAIN (Reliable Array of Independent Nodes) [5] [4] is a subcomponent for a reliable rainfinity web server. It is based on distribution of data stripes and parity among several cluster nodes. A particular data layout scheme EVEN-ODD enables the system to tolerate up to two faulty nodes/disks under an optimal redundancy/payload ratio.

PVFS [13], [8] is a striped file system for Unix clusters. The aim is to provide fast reading and writing access to out of core data for high performance applications. The data layout is similar to RAID level 0 over several nodes. At the current state, fault tolerance is not taken into account but is planned for future developments.

nfsp [14] is near to PVFS, but directly compatible to the NFS protocol. Data is stored in stripes over several cluster nodes. Parity schemes are not implemented yet, but may be added in the future.

A cluster video storage system (***CFS*** as denoted by the authors) [7] was developed at the University of Lyon on the basis of a PC cluster connected by a Myrinet network. The target system combines RAID level 0 as the local storage basis on nodes and RAID level 3 as the data layout scheme applied over different computers. Video files are gathered through a high speed system area network (Myrinet) over a direct access interface to the network. A result from a detailed analysis is that server group sizes are to be chosen in the range between three and eight nodes. Striping over a higher number of nodes may enlarge the likeliness that one out of these server is reacting too slowly and so worsens performance.

A system that is directly coupled with a video streaming server is ***DarwinCluster***[15]. It is based on replication of the entire stored content on the disks of all cluster nodes. Several instances of the Darwin Streaming Server operate on that content. It works as a plug-in module for Darwin Streaming Server and redirects rtsp connections from a master server to a server with currently lowest load. A drawback is the necessity to copy all the content from the master server node to all client nodes. Anyway, for scenarios with read-only content and sufficient disk space on the cluster nodes this is a practical solution.

Tiger [6] is a video file-server that also stripes video files among several computers with attached disks. A particular feature is that accesses to stripes are taken with a predefined schedule taking the read times and the play times into account. Stripes are replicated completely, whereby secondary copies are placed in the inner zones of the disk. So the primary copies use the part of the disk that deliver data with fast speed and the secondary copies use the slower disk zones.

Yima-1 [17] is a cluster-based streaming server that also relies on the Darwin Streaming Server for packet assembling and delivery of data. Data is retrieved from a distributed file system that stores data blocks in a pseudo–random manner distributed over the nodes. *Yima-2* uses a decentralized approach where only a rtsp control server runs on a dedicated node and the delivery of packets from the storage is performed from the nodes that store data locally. In that way, the Darwin Streaming Server had to be replaced by another streaming component. Additionally, the video player on the client side must be aware to receive packets from several nodes and to combine it to a single stream.

III. Target System

The *netRAID*-system is designed for multicomputers with local disks. A typical target system is a PC cluster. Nodes have to be connected by at least one switched network. For coupling storage systems together and for transferring data to the clients it is valuable to employ two separated networks.

On each node runs a server process, being responsible for access to the local disk. These servers accept TCP–connections from clients and perform access to data stripes on request. Data to be written or read is transferred over the network. For *netRAID*-clients, a library translates file access operations to interactions with storage servers. Interaction with the *netRAID* storage system always covers blocks of a predefined and globally equal length. To adapt file accesses to this length, buffering, splitting and assembling of data blocks is integrated into the client component. Additionally, a client component is responsible for calculating parity information and if necessary for reconstruction of missed data blocks.

NetRAID–clients are application processes that directly process stored data or process and deliver data for external usage. In particular, we applied the *netRAID*–library to the Darwin Streaming Server [10]. Many instances of the Darwin Streaming Server run on the cluster and share the load of data delivery. The file access is done over a *netRAID*–client interface. In this application scenario, separated networks for intra–cluster block transfer and for connections of streaming servers to quicktime–clients were used.

IV. Fault Model

As to the fault tolerance aspect, it is assumed that for a specific time only a single node fails. A failure includes timing failures and crashes on the level of storage node activity. A timing failure is assumed, when a storage node does not reply within a specified time. Additional faults can not be tolerated until the failed node has been recovered or replaced by another one. The latter includes that lost information is reconstructed. Although a fault of any single node can be tolerated by the storage scheme, the cases where a requesting process fails during accessing the distributed data are not covered. This has to be handled by techniques on a higher level (e.g. redundant server processes with IP failover). Additionally, crash and timing failures of clients are taken into account. The network is assumed to be fault-free. Either, the network can consist itself of totally redundant levels or can be assumed to be much more reliable than the storage nodes.

V. Distribution Scheme

The data layout scheme according to RAID level 3 is illustrated in figure 1. A group of server nodes form a parity group. In such a parity group, blocks of a predefined length are split in striping units and stored on several nodes. An additional node stores the parity information that is derived from striping units within the parity block. Parity information expresses a bitwise XOR-result over these stripes. On the side of storage servers, striping units of a data object are stored sequentially in a file using the local file system. Naming of the server files is done by a concatenation of the data objects name with the identification number of the storage server.

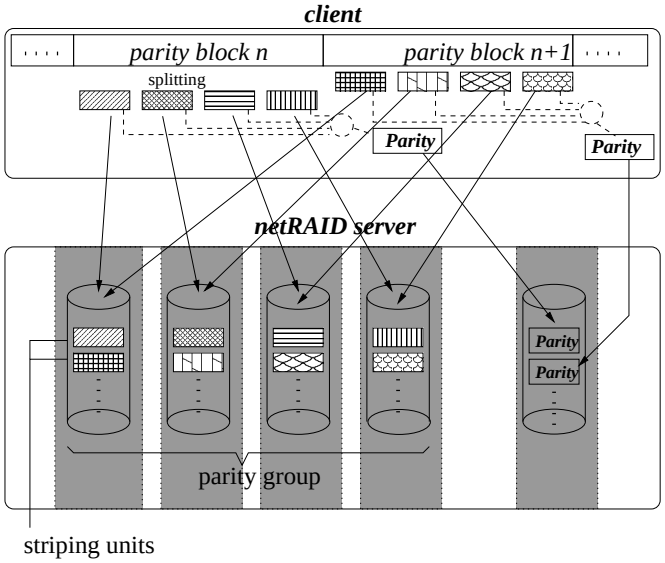

Fig. 1. RAID Level-3 Data Layout

When a storage node fails, only a single striping unit of a parity block gets inaccessible. This case is detected by missed replies from the failed server and the missing striping units are reconstructed using parity information. The missed block is derived by a bit wise XOR-ing of all available striping units and the parity information. This is done in the client.

Incompletely filled parity blocks, normally occurring at the end of data objects, assume zero bytes for unused space in the context of parity calculation. In our implementation, additionally to the parity the used length of the parity block is stored on together with the parity information. This guarantees to obtain the correct data object size, even in the case that a striping unit is lost that covers the end of a data object. Implementing such a storage scheme over several nodes is relatively easy, whereby the storage scheme itself is well understood. Still a problem is the occurrence of concurrent read and update operations within a parity block. In that case, a read operation may gather striping units that represent a different state, some may be already updated and some others may still contain old content. The same problem arises when several clients update blocks concurrently. Then striping units may represent content of different update operations. Gathering the single striping units over the network would then lead to invalid content. Especially, when data is coded in a particular form (compressed files, encoded video) the entire data may become useless. This problem can be solved using a relatively simple technique as described in the next section.

VI. Consistency

To address the problem of conflicting access to striping units, a technique can be used that is already known from databases. In databases, distributed transactions have to alter tables on several nodes, concurrently to reading accesses. For these transactions data consistency has to be ensured, in a way that either all table entries reflect a state before updating or after the update. This consistency problem is similar to updating all striping units of a parity block during concurrent read access. One solution is to order read and update operations in a way that no corrupted data can be read. In context of database technology, this technique is called time stamp ordering [2]. The proposed solution for RAID-like storage schemes is a particular application of this well known solution. The concept of that ordering scheme was first published in [16].

The ordered access is activated only in case of clients accessing the same data object during the same time period. For that decision, for each data object an object identification number is calculated by hashing over the data objects' names. If for a particular access sequence another access sequence with the same identification number is present, the ordering is activated, otherwise no ordering is necessary.

A. Ordered Access

On storage nodes, so called access instances execute read and write operations to the local file system. Their activity is driven by events – incoming request messages and self-induced events. Ordering consists in either immediately fulfilling the request or in postponing the needed activity until conflicts will disappear. For that access control, each access sequence p is assigned a tuple $(lsn_p, access\text{-}mode_p, v_p, state_p)$ in the memory of access instances. The first element is a number *lsn*, which is uniquely chosen for an access sequence. An access-mode $\in \{R, U\}$ indicates whether it is a sequence of read (R) or update (U) operations.

The number v specifies the parity block number within the last activity occurred for the particular access sequence. For internal coordination, a state $\in \{WAIT, ACC, FIN\}$ has to be stored additionally. This state expresses if the access has to wait for another ($WAIT$), if the server is currently accessing the striping unit (ACC) or if the access is finished and no further request was received for accessing the particular data object (FIN).

The basis of the ordering mechanism is the lsn, a number that has to be determined for all storage nodes equally and the block number v that determines the access position in the local sequence of stripes. With an open request an atomic broadcast protocol is executed that provides ordered delivery of messages. Such an atomic broadcast is a building block in distributed fault-tolerant systems and implemented by a middleware (e.g. ISIS/HORUS [3]). According to the order of received broadcast messages, the *lsn* is assigned to access-sequences.

Conflicts to a current request $(lsn_p, access\text{-}mode_p, v_p, state_p)$ that would violate consistency are characterized as follows:

> **Case 1:** An entry with $lsn_i < lsn_p$ and $access\text{-}mode_i = U$ is found and one of the following conditions holds:
>
> - C1: $v_i \leq v_p \wedge state_i \neq FIN$
> - C2: $v_i < v_p \wedge state_i = FIN$
>
> In this case, an earlier started sequence of write accesses has not yet updated the striping unit that is requested to be read or written.

Additionally, if $access\text{-}mode_p = U$, it has to be checked for a second case:

> **Case 2:** Another access sequence with a lower $lsn_j < lsn_p$ and $access\text{-}mode_j = R$ is present and C1 or C2 applies (see Case 1).
>
> Here, the current write request would overwrite a striping unit that has not been read in a previously initiated sequence of read accesses.

In both cases, the current access associated with $(lsn_p, access\text{-}mode_p, v_p, state_p)$ must be postponed. So, the data is tentatively not accessed and the $state_p$ is set to $WAIT$.

Figure 2 illustrates the states that are assigned by the access instance to a sequence of accesses. A sequence of access operations always has to start with an open-request. Here, the lsn is determined and a newly created table entry with $v = -1$ and $state = FIN$ is created.

In case of no conflicts, requests lead to the state ACC, whereby v is incremented and the access is immediately fulfilled. When the access is finished, a response message is sent to the requesting client and the state is set to FIN. Each fulfilled access is triggering a reactivation event. If conflicts are present for an incoming request, v is incremented and state is set to $WAIT$. For a write request, data that can not be written immediately must be buffered. Waiting requests are reactivated, when other accesses have finished and conflicts disappear.

A reactivation event lets the access instance check waiting requests in ascending order of their *lsn*. The first found request (represented by a table entry) that is not in conflict with other

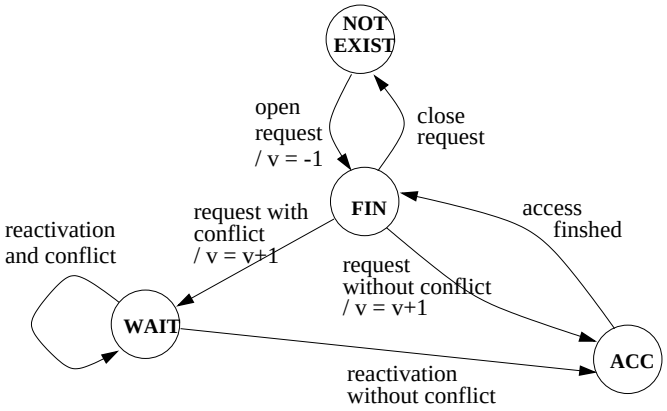

Fig. 2. States and transitions for access ordering

access sequences is reactivated and its state is set from $WAIT$ to ACC. This is followed by sending back a response message and changing the state to FIN. After a reactivation and a fulfilled request, a new reactivation event is initiated.

The described scheme assures that each striping unit is accessed by clients in the order of ascending *lsn* and thus on each node in the same order. The table entries *(lsn, access-mode, v, state)* can be found in figure 3 whereby the position of entries is related to a virtual time-line of accesses. This vertical position expresses the value v in a graphical representation. Entries in state ACC are directly positioned in the striping unit v, in $WAIT$ are placed above the striping unit and those in FIN below. In that representation

1) time lines of a pair of reading access sequences are allowed to intersect.
2) if at least one access of a pair is a write operation, the time lines are not allowed to intersect.

The reason for (1) is that read accesses do not influence each other and therefore do not need to be ordered. In figure 3, among read sequences with *lsn=4* and *lsn=5*, no ordering is needed. In the example, the write sequence *lsn=1* needs an ordering related to all other access operations. In the used graphical representation, each related pair of points of these timeliness is kept in the same relation to the other ones. Only when an updating client is present (2), ordering is activated.

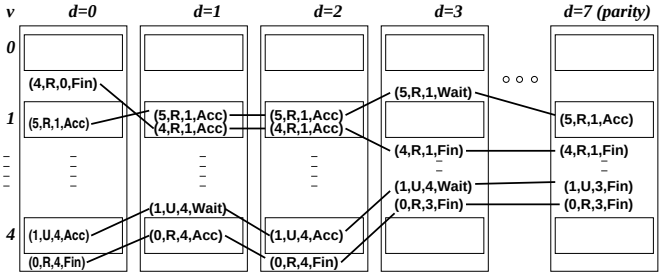

Fig. 3. Snapshot of sub-operations on different stripes

B. Side Effects

Up to this point, we focused on consistency in scope of a parity block access. Ordering ensures that a single access

is free of any corruption due to concurrent activity of other clients. Beyond this scope, such an ordering scheme ensures that several clients get sequential access to succeeding parity blocks in the same order related to each other. So, an implicit coordination is given, also for sequences of accesses. A reading client that starts access after an updating client will always get the updated data portions when invoking block-wise read operations sequentially. Accordingly, if two writing/updating clients access the same data object sequentially, one particular client will always determine the last written data portion. This ensures that a data object is left in a consistent state. Consistency is reached at the expense of dependency of operations in terms of speed. Thus, a negative aspect is that accesses may get blocked until data is accessed by other clients. For instance, a writing client must wait, in order to avoid overwriting of data that is still to be read. A slow reading client may slow down the writing process. Effects of faulty clients are analyzed in VI-C.2.

C. Fault Tolerance of the Ordering Scheme

Ordering introduces dependencies among different accessing clients, if accesses are related to a common data object. So, beyond data availability under faults, the ordering must be checked for proper function under faults.

1) Faulty Server Nodes: The question is, how this ordering on a faulty acting storage node may affect consistency. As mentioned in IV, timing and crash faults of a single node are to be tolerated.

Potentially, ordering could get violated in two ways. The first potential fault scenario (i) would be an access that is executed without waiting for possibly existing accesses that must be executed before. The impossibility of this scenario can be explained by following two points:

- When all present accesses are represented by a valid *lsn* and a table entry, such a behavior (i) can not occur due to a crash or a timing fault. Solely, a current request may be ignored due to a crash or get delayed by a timing fault.
- Case (i) would occur only, when an access sequence with a smaller lsn is not represented in the local table. A crash or a timing fault can not cause this situation without omitting the table entry for the present access operation too. Because a current request can not get effective without an entry in the servers table for the particular *lsn*, this scenario is impossible.

The other potential fault scenario (ii) is an access that is set to $WAIT$ when no dependency to other accesses is present.

- Such a situation may occur, for instance when another access is finished and the node crashes before setting the state to FIN. Then the node is crashed for the present access too. The same applies for a timing fault. If the access management is delayed, also the processing of new requests gets delayed. This is equivalent to a not responding storage server.

So, there is no way for order violation, even in the case of faults (that comply to the fault scenario). The only remaining effect is that a node does not process an access at all or only incompletely and then is unavailable for all subsequent access requests. This behavior is tolerated by the storage scheme and the fault correction at the client side.

2) Faulty Clients: As described in VI for consistency reasons, it is necessary to order accesses of several access sequences that are initiated by several client processes on different nodes. A drawback is that these dependencies cause a vulnerability of the server system against faulty acting clients. According to the fault model, also a client may crash or delay accesses due to timing faults. In both cases,

- a client that initiates a sequence of read accesses and then fails, causes blocking of concurrent write accesses.
- a failed writing client will block concurrent read and write requests.

Because a client sends requests to many server nodes, it will be possible as well that not all messages are sent and the client fails or delays activity in a timing fault manner. Such an inconsistency may occur which is known as a partial distribution fault. In such a scenario, a subset of servers can be get blocked for dependent accesses.

The basic ordering scheme would block accesses with a larger *lsn* for all parity blocks that were not yet accessed by the failed client. This means that a single crashing client may block the access to a particular data object on all server nodes. This weak point has to be avoided by an extension of the basis scheme.

The extension consists in an assertion of a maximum access interval $intv_x$. If a client does not invoke the next access within $intv_x$, the ordering related to the particular access sequence is deactivated. Other accesses proceed in case of conflicts, but the clients are informed about possible consistency problems. In the following we use t as the current time and $t_{access,x}$ as the time of the recent access of access sequence x.

So case 1 as described in VI-A has to be restricted in the following way:

> **Case 1:**
> Another entry with $access\text{-}mode_i = U$ and a lower $lsn_i < lsn_p$ is found and C1 or C2 applies. Only, if $intv_i \geq t - t_{access,i}$ holds, the current access has to be postponed, according to the basic scheme.
> If a conflicting access sequence is present that does not comply with the predefined maximum access interval, $intv_i < t - t_{access,i}$, then the current access can get activated, if no other conflicts apply. The client has to be informed that the current access is influenced by a delayed concurrent operation. This is done by responding with a notification of active consistency violation (ACV). Access sequence i is marked in the server data structures as being influenced by a passive consistency violation (PCV).

Case 2 that is effective when the current access is an update request ($access\text{-}mode_p = U$), must be altered in the following way to comply to an environment with faulty clients.

> **Case 2:**
> Another access sequence with $access\text{-}mode_j = R$ and a lower $lsn_j < lsn_p$ is present and C1 or

C2 holds. According to the basic scheme, then the current write request is conflicting with the found one.

A cause for postponing the current request is only given, when $intv_j \geq t - t_{access,j}$ holds. But if the conflicting access sequence is delayed longer than the allowed interval ($intv_j < t - t_{access,j}$), the current access is allowed to be executed (when no other conflicts forbid this). The entry for $access_j$ has to be marked as a passive consistency violation (PCV). For the current request, no consequences in terms of consistency are present. But, if the delayed access sequence is reactivated again, it will respond with an PCV notification. This situation can be handled on application level.

Because of the decoupling of dependent accesses that do not comply the timing constraint, a reactivation event has to be induced by the system itself, after a time of $min(intv_x)$ relative to the last reactivation event.

When a client receives a response with an order violation notification from only one server f, it can mask that fault. A reading client can reconstruct data from server f using the parity information and deliver data to the application. If a writing process gets an ACV response, it proceeds operation. If PCV is reported on more than one server, a reading client must assume data to be corrupted. The reaction is application–dependent and may range from application stop to just ignoring the corruption and continuing reading and processing. In principle, it must be assumed that the data is corrupted by a concurrent write operation. A write sequence with an ACV must proceed, whereby in case of a PCV, a write sequence gets obsolete and can be stopped.

VII. Performance Analysis of the Storage System

A requirement to the distributed server system is the ability to upstream and downstream video data fast enough, even when many users are downstreaming content concurrently. We analyzed the performance of the storage system from the viewpoint of many streaming servers that are acting as clients of the storage system. These streaming servers run inside the cluster concurrently to the activity of storage server processes. Such a client may use two different strategies for data access:

Sequential access – Striping units are requested and received one after another. After requesting a stripe, the client waits for a response from the storage server.

Multithreaded access – All striping units are requested without waiting for response messages. Threads are used to receive the data from the storage servers. From the perspective of a client the data is gathered in parallel.

A. Measurement Setup

We use an eight node pc cluster as video server. Each node is a double-processor Intel Pentium III system, and contains an IDE disk (Seagate ST 330621A). Storage nodes run under Linux and local data storage is accessed by the reiser-fs file system. There are two types of networks present, a 100 MBit/s switched Ethernet and additionally a Gigabit Ethernet network (GE). For coupling the storage nodes we exploited the fast GE network.

B. Single Accessing Client

First we measured the obtained throughput by using a single process that reads or writes on the *netRAID* system. We varied the number of involved storage nodes (striping group size) and the granularity of striping (length of striping units). In the experiment, a 300 MByte video file was used. In the single client scenario, a multithreaded access strategy has been used. Figures 4 and 5 show the results. A maximum read throughput can be found on a striping unit size of 24 KByte and a number of seven storage servers. The maximum write performance was found with eight server nodes and a striping unit size of 16KByte.

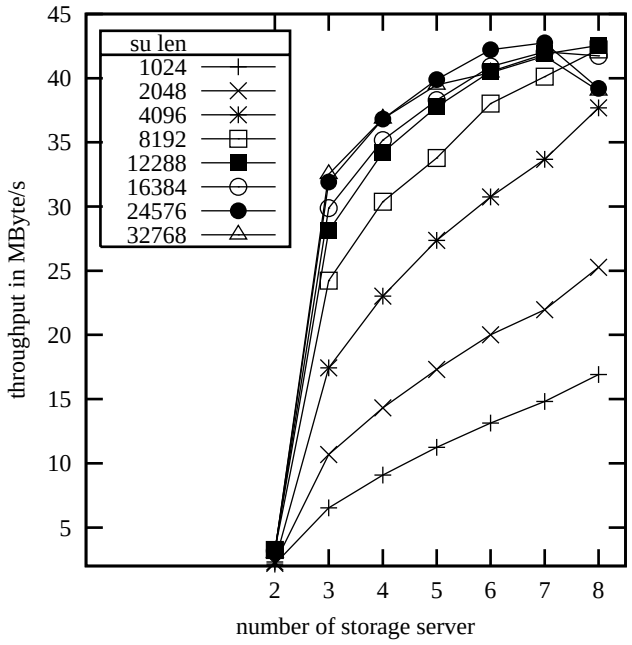

Fig. 4. Throughput measured for a single downstreaming client. The measurements reflect different striping unit sizes (see legend).

C. Many Reading Clients

In a second step, we measured the obtained throughput by many processes that read data objects from the *netRAID* system in parallel. In order to start read access sequences of several clients nearly at the same time, a controller process was used that sends a start message to clients and finally receives the number of read bytes from each client. This controller was also used to obtain the total throughput as the sum of read bytes over all clients in a time that has expired from sending the start message to the reception of the report message from the last client.

Figure 6 shows results for a parity group size[1] of 7+1 and a striping unit length of 24 KByte. One scenario (denoted by

[1] 7+1 denotes a group of eight storage server, whereby one server acts as parity storage.

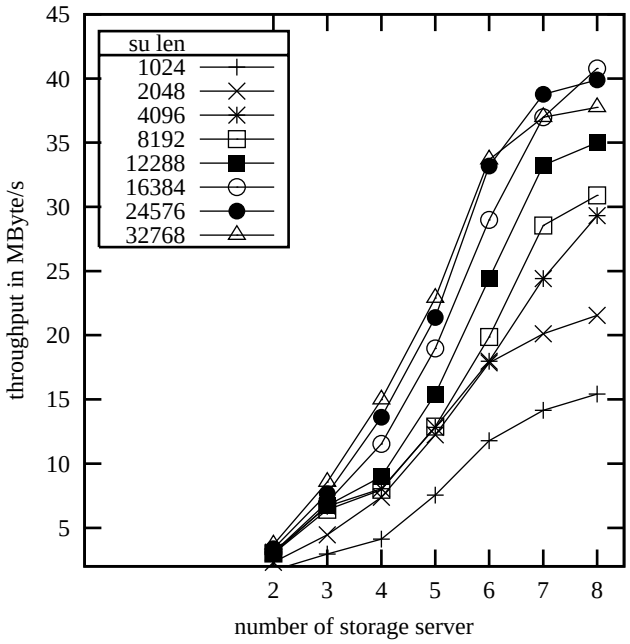

Fig. 5. Throughput measured for a single writing client.

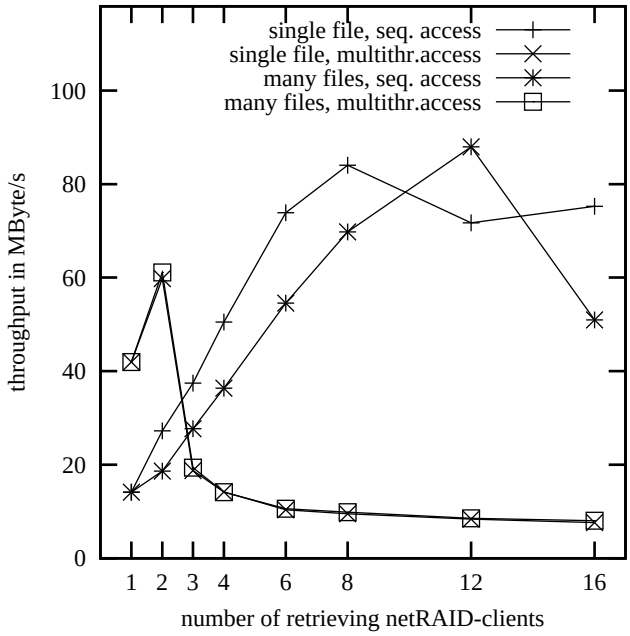

Fig. 6. Total throughput measured for several downstreaming clients.

single file) is given by many clients which are accessing one common file and a second scenario (denoted by many files) covers access to different files.

For multithreaded access strategy, a maximum throughput is observed when two clients are accessing stripes concurrently. A higher number of clients leads to a performance degradation. With more than two clients, the sequential access strategy provides a higher throughput. The reason for that is the high number of threads in the multithreaded access scenario, which is only beneficial when a single or a few *netRAID*-clients run on a host.

D. Concurrent Read/Write

Figures 7 and 8 show results for a 7+1 data distribution, whereby one half of nodes are writing to the storage system concurrently to clients that read data. We distinguish following scenarios:

- 1r+1u One client performing a read operation and one client updating a data object is present.
- 2r+2u Two client processes are performing a read operation on two different data objects and two clients are updating data objects.
- 4r+4u Here, four data objects are read by four clients in parallel. Additionally, four clients are updating data objects.

The first part of the experiment is based on independent accesses, so that the read and the written/updated data objects are different ones. The ordering is not activated. The second part assumes common data objects. In all scenarios one reading client and one updating client use a particular data object in common. So, for all accesses the ordering is activated.

In figure 7 and 8 the values represent the sum of all access rates, including read and write. Each diagram can be used for a comparison between dependent and independent access. The total data size that has been read and written by all active clients has been measured over a period of time starting with the activity of the first client and ending when the last active client has completed access. In such a way, scenarios where slower clients are longer active than other faster clients are covered. The results show that there is no performance degradation due to ordering when clients access data sequentially in a best effort manner.

Multithreaded access led to the effect that ordered accesses are slower than independent accesses, but absolutely exhibit higher access rates (in cases 1r+1u and 2r+2u) compared to sequential access.

E. Discussion

The reported performance results for single client processes are comparable with other systems (e.g. [14]), so that we argue that the ordering of accesses does not influence the performance significantly. Additionally, by activating the ordering (comparing results of VII-D), we could not notice a significant performance degradation. Moreover, when accesses are dependent (related to same date objects), data can be more often found in disk and file system caches. Thus, in some cases, dependent access sequences will reach a higher total throughput than independent access sequences.

VIII. Further Development

Concurrent access in *netRAID* demands further optimizations. One point is reading in advance and caching in the storage server. When a high number of accesses are present on the storage server, a more intelligent scheduling of accesses promises an improvement. For instance, non dependent accesses may be scheduled bulk-wise in order to employ locality and to

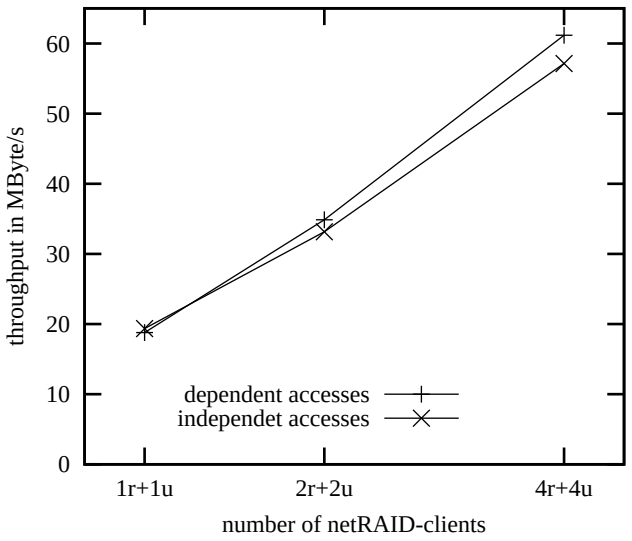

Fig. 7. Sequentially accessing clients: comparison between independent and dependent accesses.

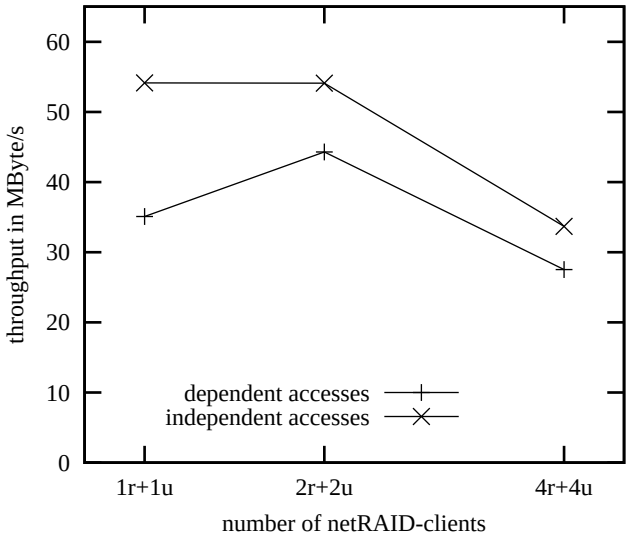

Fig. 8. Multithreaded accessing clients: comparison between independent and dependent accesses

minimize accesses to different disk regions and to utilize caching in a better way.

As mentioned before, the *netRAID* system is coupled to the Darwin Streaming Server. For that, several streaming server run on several cluster nodes. For this application level, a middleware for load balancing and fault tolerance among these servers is currently developed. It consists in a group management with built–in diagnosis coupled with a global exchange of load information.

IX. Summary

A distributed storage system for multicomputers with local disks, called *netRAID*, has been described. Particularly, we focused on a method for a guaranteed consistency for concurrently up- and downstreaming clients in an adopted RAID-like distribution scheme. The solution consists of ordering of access operations, according to an order determined on the start of each access sequence. So, coordination among several storage nodes takes place at the time of beginning a new sequence of access operations to a data object. During accesses no additional communication is necessary. The local ordering is feasible, because only the order among write and read accesses determines the output. Experiments show that the ordering by itself does not influence the performance significantly. But causal dependencies are introduced - access operations possibly have to be blocked until other dependent operations are finished. These may slow down accesses in particular cases. This side effect comes together with an extended consistency for concurrent sequences of access operations on sequential structured data objects and can be used to update content during access without data corruption.

References

[1] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless Network File Systems. In *15th Symposium on Operating System Principles, ACM Transactions on Computer Systems*, 1995.

[2] P.A. Bernstein, V. Hadzilacos, and N. Goldman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, 1987.

[3] K.P. Birman and R. Van Renesse. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, 1994.

[4] M. Blaum, J. Brady, J. Bruck, and J. Menon. EVENODD: An Efficient Scheme for Tolerating Double Disk Failures in RAID Architectures . *IEEE Transactions on Computers*, 44(2), February 1995.

[5] V. Bohossian, C.C. Fan, P.S.LeMahieu, M.D. Riedel, L. Xu, and J. Bruck. Computing in the RAIN - A Reliable Array of Independent Nodes. Technical report, California Institute of Technology, September 1999.

[6] W. Bolosky, J. Barrera, R. Draves, R. Fitzgerald, G. Gibson, M. Jones, S. Levi, N. Myhrvold, and R. Sashid. The Tiger Video Fileserver. In *International Workshop on Network and Operating System Support for Digital Video and Audio*. Springer, 1996.

[7] A. Bonhomme and L. Prylli. Performance Evaluation of a Distributed Video Storage System. In *IPDPS 2002, Proceedings (CDROM)*. IEEE Computer Society, 2002.

[8] P. H. Carns, W. B. Ligon, R. B. Ross, and R. Thakur. PVFS: A Parallel File System for Linux. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, 2000.

[9] E.N. Elnozahy. Storage Stratigies for Fault-Tolerant Video Servers. Technical Report CMU-CS-96-144, Carnegie Mellun University, 1996.

[10] Apple Computer Inc. "QuickTime Streaming Server 4.1. Datasheet on http://www.apple.com/quictime/products/qtss, 2002.

[11] Tivoli Systems Inc. Vision – Tivoli Storage Management Solutions for the Information Grid. White Paper, 2000.

[12] R. Katz, G. Gibson, and D. Patterson. Disk System Architectures for High Performance Computing. In *Proceedings of the IEEE*, pages 1842–1858. IEEE Computer Society, December 1989.

[13] W. B. Ligon and R. B. Ross. An Overview of the Parallel Virtual File System. In *Proceedings of the 1999 Extreme Linux Workshop*, June 1999.

[14] P. Lombard and Y. Denneulin. nfsp: A Distributed NFS Server for Clusters of Workstations. In *IPDPS 2002, Proceedings (CDROM)*. IEEE Computer Society, 2002.

[15] Darwin Plugings. Clustering/Load Balancing for Darwin Streaming Server, http://www.darwinplugins.com/darwincluster. 2002.

[16] P.Sobe. Concurrent Updates on Striped Data Streams in Clustered Server Systems. In *IPDPS 2001 Proceedings (CDROM), Workshop on Fault-Tolerant Parallel and Distributed Systems*. IEEE Computer Society, 2001.

[17] C. Shahaby, R. Zimmermann, K. Fu, and S.D. Yao. Yima: A Second-Generation Continuous Media Server. *IEEE Computer*, pages 56–64, June 2002.