

Issues of Many-to-one Communication in a Distributed RAID *

Alessandro Di Marco, Paolo Gianrossi, Giuseppe Ciaccio
DISI, Università di Genova
via Dodecaneso, 35
16146 Genova, Italy
{dmr,gianrossi,ciaccio}@disi.unige.it

DISI Technical Report DISI-TR-05-14

13 October 2005

Abstract

Any set of autonomous workstations, however networked (by a LAN, a MAN, or wireless), can be seen as a collection of networked low cost disks. Such a collection can be operated by proper software so as to provide the abstraction of a single, larger block device, made available to all the participants on a peer-to-peer basis. By adding enough data redundancy, the disk collection as a whole could act as single distributed RAID, providing capacity and reliability along with the convenient price/performance typical of commodity hard disks.

This paper reports the latest achievements with a prototype of distributed RAID device called DRAID. DRAID offers storage services under a Single I/O Space (SIOS) block device abstraction. The SIOS feature implies that the storage space is accessible through each of the participant stations, rather than through one or few fixed end-points. The paper focuses on the issues concerning the many-to-one communications arising when a participant host reads data which are striped across a number of other participant nodes.

Keywords: RAID, network-attached storage, peer-to-peer, parallel I/O, single system image.

1. Introduction and motivation

The information age we live in poses increasingly hard requirements on storage systems. The need for fast and reliable storage is imperative in many mission-critical frameworks, from corporate to medium-size enterprise to scien-

tific experiments, in which data rate requirements in the range of GByte/s are often coupled with very strong availability assumptions. But also the cost of the processing and storage infrastructure is a critical parameter.

At the low end of the spectrum of storage technologies we still find conventional EIDE disk drives. Such low cost commodity devices have shown tremendous improvements in terms of cost/capacity, but on the other hand their transfer rate has grown very little in comparison to other system components. In a typical modern PC all data transfers among distinct subsystems take place at a speed in the GByte/s range, with the very exception of EIDE disks, the peak transfer rate of which lags behind a modest 100 MByte/s, not to mention the known lack of robustness and fault tolerance of such entry-level devices.

At the high end of the spectrum, however, we find storage devices providing higher capacity together with much greater performance, thanks to techniques known as data declustering and disk striping [17, 15] which allow some amount of parallelism across many disks. A four-way SCSI disk array can yield a transfer rate in the order of a GByte/s. However, incorporating a large number of disks into a disk array makes the storage system more prone to failure than a single disk. Higher reliability is then achieved by using redundant encodings of data so as to survive one or more disk failures, yielding what is called a RAID [12]. The main disadvantage of a RAID is a far worse cost/capacity aspect (with the exception of the entry-level RAID-1 systems found on small workstations), compared to EIDE or SCSI disks. Another disadvantage comes from the centralization of multiple storage devices into a single package, which greatly increases the stochastic correlation among single disk failures. The centralization also forces all disk accesses to be physically conveyed throughout a limited and

*This research is supported by the Italian Ministry of Research, FIRB grant RBAU01EK3L

predetermined number of access interfaces, which could act as hot spots.

An alternative and more convenient answer to the need for collective, high-capacity and reliable storage, viable in a typical corporate or academic environment, is to aggregate the huge amount of unused disk space found at the various workstations and PCs in use. Indeed, a RAID could also be implemented according to a distributed organization in which the storage capacities of individual networked hosts are logically aggregated and then shared among all the participants. Such a *distributed RAID* [19], or “Do-It-Yourself RAID” [2] can provide a capable, dependable, and cost-effective storage system made out of all the unused and inexpensive hard disks provided by the participants. In a distributed RAID, a proper distributed software running on the participant hosts manages all individual disk requests. The system manages data striping and replication over the disks provided by the participants, according to some predetermined mapping, using the existing networking infrastructure for transmitting the data.

A software RAID allows arbitrarily sophisticated techniques for data striping and redundancy, that is, a far greater flexibility in trading speed and space for reliability. A distributed RAID is easily expanded by just adding more participant hosts. The peer-to-peer organization of a distributed RAID yields greater availability, thanks to the absence of single points of failure. The dispersion of participants across a wider area leads to a lower correlation among disk failures. Last but not least, this approach tends to preserve the favourable cost/capacity aspect typical of commodity hard disks.

The early works on distributed RAIDs [19, 2] only focused on the opportunity for better cost/capacity. Only a decade later it emerged that these systems could offer another, important advantage over classical RAIDs. Indeed, the abstraction of a single block device out of a collection of physically distinct disks is provided by cooperation among the involved stations. If such cooperation takes place on a peer-to-peer basis, each involved station can be given equal access to the common abstract device regardless of the location of physical storage resources, a feature now called *single I/O space* (SIOS) [7] which is a special case of a more general feature called *single system image* [14]. Now, providing SIOS in a distributed RAID has an important implication, namely, it allows concurrent access to the shared virtual disk through multiple interfaces, as all stations can actually serve as end-points to the virtual device. This in principle allows a potentially higher aggregate throughput and a better load balancing, compared to a classical network-attached storage system.

On the other hand, the distributed and physically dispersed organization, and the possibility for every node to act as end-point to the virtual device, pose specific chal-

lenges, namely, multiple concurrent writes to shared blocks, coherency among the local block caches of each participant host, tolerance to transient as well as persistent network failures, and potential network congestion when reading data which are striped across a number of remote participant hosts.

In this paper we report about a prototype of a distributed RAID device called DRAID, in which the challenges posed by concurrent writes and network fault tolerance have been successfully faced, and focus in greater detail on the issues raised by the *many-to-one* communication patterns taking place when a participant host, acting as a client, reads data which are striped onto a number of other participants.

2. DRAID: A Distributed RAID based on Reed-Solomon

DRAID is a virtual block device that distributes blocks across a set of networked workstations, by first computing a Reed-Solomon redundant encoding of each block [13], then striping the resulting data among disks (or any other block device) local to the participant stations.

Let us consider two natural numbers N and K . DRAID leverages a number W of workstations, logically arranged into a two-dimensional array that we call the DRAID *matrix*. Each matrix row must count exactly $N + K$ stations. It is thus required that W be an integer multiple of $N + K$.

On write, each logical block B of the DRAID block device is partitioned into a sequence of N *segments*, $B_1 B_2 \dots B_N$, which is then extended with additional K redundant segments $P_1 P_2 \dots P_K$ computed by a Reed-Solomon encoder. All the B s and the P s segments have same size. The resulting *data stripe*, $B_1 B_2 \dots B_N P_1 P_2 \dots P_K$, counting as many as $N + K$ segments, is striped across $N + K$ stations/disks in a single matrix row during the write operation. The choice of which matrix row is to store the data stripe is done by the DRAID address translation algorithm, a simple function of parameters N , K , and disk capacity C .

A subsequent read operation on the same logical block only needs to get *any* N out of $N + K$ segments from the data stripe, in order to be able to successfully deliver the entire logical block $B_1 B_2 \dots B_N$. During normal operation, preference is given to the first N segments of the data stripe, as they do not require any Reed-Solomon calculation to deliver the original block. The redundant segments are taken into account only in case up to K of the first N segments become unavailable due to one or more faults; in this case, the original data are obtained by Reed-Solomon calculation over the available segments.

A realistic distributed RAID might count quite a lot of disks. DRAID can be easily expanded at run time by hot-plugging new hosts in multiples of $N + K$ and putting them

as new rows in the matrix, so that existing rows do not change size or composition.

The current prototype of DRAID is operated by a peer-to-peer software in the form of a kernel module for Linux 2.6. It does not require modifications to the kernel. The DRAID module works as a set of Linux kernel threads. Such multi-threaded architecture allows multiple outstanding communications to be in progress at the same time, thus achieving a better overlap among them as well as with other DRAID tasks. Allowing multiple outstanding communications is crucial for performance, as each request/response cycle may require a long completion time.

Following a peer-to-peer approach, both client and server functionalities of DRAID are implemented by the same module. This allows DRAID to provide a SIOS abstraction, as each participant host has the same logical view of the entire virtual block device regardless of its own particular placement in the matrix. Moreover, each host can act as an independent end-point to the logical block device; this provides a potentially higher aggregate I/O bandwidth, prevents hot spots, and also makes DRAID suitable to mobile clients and parallel I/O tasks.

The DRAID communication layer is based on UDP. UDP datagrams are stamped by sequence numbers, to allow each DRAID receiver to detect missing datagrams and ask for retransmission. Usually a DRAID message is made of a single UDP datagram, so the retransmission is asked for the entire DRAID message; for simplicity, this policy is applied to all DRAID messages, including the multi-datagram ones. No congestion control is provided.

3. Managing failures and concurrent writes

If a given station W_i in a DRAID row has a failure while a client is writing a segment of data to it, a relocation rule in the client diverges the segment to a temporary storage area, located on an alternate node called a *backup* host. Completing the write, although on a possibly different host, is crucial in order not to alter the amount of data redundancy. The backup host is chosen at a different matrix row, but in the same *column* of W_i . The relocation rule also allows to find the backup station and thus the diverged segment during subsequent reads, should the failure at W_i persist.

If the failure was transient and W_i resumes operation, a clash could arise between the relocated segment and the obsolete copy still stored at W_i . The clash is avoided by versioning all the segments of a data stripe on write, then performing a check on read to detect possible old-versioned segments. When an old-versioned segment is found, it is immediately replaced by up-to-date data coming from the corresponding backup host, and normal operation is restored.

The segment versioning also allows to manage concurrent writes on a given block. Having two stations A and B writing different content to the same logical block of DRAID might result in a write interleaving over the segments of the data stripe, producing a hybrid stripe with incorrect content. By versioning all segments in a data stripe before writing, and then testing the current version number after writing, both A and B are allowed to detect a clash. The two stations resolve their contention by each increasing their own current version number by a random quantity, then writing the block using their respective updated version number, and repeating until no clash is detected any more.

4. Improving performance: write clustering and read ahead

Communication performance is notoriously poor when short messages are sent. Messages carrying data blocks from/to disks pay an even larger penalty, due to the huge latencies of disk operations. As a consequence, acceptable levels of performance can only be attained by clustering together those block requests which happen to be contiguous in the address space of the virtual block device. We call a *block cluster* any sequence of contiguous blocks in a block device.

On write, the current prototype of DRAID supports *write clustering*. The Linux buffer cache manager reorders the virtual blocks written in the buffer cache, so as to maintain them ordered by address. When the buffer cache is full, a kernel thread runs which flushes the cache to the physical device. This thread cooperates with the low level driver of the physical device. With DRAID, the low level driver manages the blocks evicted from the buffer cache by identifying convenient block clusters then acting upon these.

Let us suppose our DRAID be arranged into a matrix with rows of $N + K$ disks each. Each block in a block cluster can be encoded by Reed-Solomon as a redundant data stripe formed by $N + K$ segments, $S_1 S_2 \dots S_{N+K}$. So, the block cluster can be turned into a cluster of data stripes. By dividing each stripe into its segments then grouping together homologous segments, we turn the cluster of stripes into $N + K$ clusters of segments, each one to be written to a distinct disk. At this point, the DRAID module issues $N+K$ “cumulative” write operations, each directed towards a distinct disk. Each write operation requires an acknowledgement from the destination station/disk, if remote; the latency of acknowledgements is hidden by allowing multiple in-flight write operations.

By design, contiguous DRAID virtual blocks result in contiguous data stripes which are kept contiguous on the physical disks. Write clustering thus optimizes both on network communications and on physical disk writes.

On read, the Linux buffer cache manager implements a different optimization, namely a *read ahead*. Each read operation to a given virtual block B_l actually triggers reading a whole sequence $B_l, B_{l+1}, B_{l+2}, \dots, B_{l+r-1}$ of r consecutive blocks, where r is a parameter of the Linux kernel. This is a read of a block cluster, and the low level driver of DRAID accomplishes this by issuing $N + K$ “cumulative” requests, each one directed to a distinct hosts. This way, the number of distinct read requests is independent from the block cluster size.

5. Performance testbed and microbenchmarks

We have measured write and read performance of the bare DRAID virtual block device, with no file system on top of it. The block size of the DRAID virtual block device was set to the maximum value allowed by Linux 2.6, namely, 4 KByte.

The measurement testbed is a cluster of dual-CPU PCs, each with two Xeon 2.8 GHz processors, 1 GB registered ECC RAM, a Maxtor Diamond Max Plus 9 80 GByte ATA disk, and an Intel PRO/1000 Gigabit Ethernet adapter hardwired on the Tyan Tiger i7501S motherboard. The PCs are networked by an Extreme Networks Summit 7i Gigabit Ethernet switch, with copper wiring.

In order to measure the time spent in writing N blocks, we run the UNIX command `time dd if=/dev/zero of=/dev/draid count=N` on one of the nodes of DRAID. Read performance was evaluated by running the command `time dd if=/dev/draid of=/dev/null count=N`. These simple microbenchmarks catch the DRAID behaviour in the case of long sequences of contiguous blocks, which is representative of a practical use of DRAID for reading/writing large files. We do not expect any distributed RAID to yield good performance with “small” reads/writes, because of the latency incurred by too many parts of such a system (disk seek, communication, thread scheduling, Reed-Solomon encoding). In our measurements, the elapsed time includes the Reed-Solomon encoding time, and the total amount of read or written data comprises the redundant part.

Some of the reported measurements only concern the communication subsystem of DRAID in the case of read operations. In this case, to mimic the typical pattern of a read operation, we have run a kind of revisited “ping pong” microbenchmark in which a requestor P sends empty messages towards a set of responders, which then send non-empty answers back to P. P measures the time interval T between the emission of the first request and the arrival of the last answer, and computes the communication throughput as S/T , where S is the aggregate size of the collected answers.

In all experiments, each block of data is striped onto

eight nodes, two of which are redundant. The entire cluster is thus arranged in rows of eight nodes each. Striping degrees in excess of eight do not pay, as the Gigabit Ethernet interconnect seems to saturate or even congest when a host reads more than eight stripes simultaneously from as many nodes.

6. DRAID performance on read

To read any given DRAID virtual block, a given DRAID station P issues $N + K$ read requests to other DRAID hosts in a short time interval. Shortly after, as a consequence, P gets a burst of datagrams coming from as many remote hosts. Such a *many-to-one* communication pattern can cause a network congestion.

In search of solutions to such congestion problem we have played with the different tactics accounted below.

6.1. First tactic: best-effort with limited read ahead

Figure 1 shows the DRAID asymptotic read throughput as a function of the degree r of read ahead, that is, the number of consecutive 4 KByte blocks collected by each single read operation.

The positive effect of read ahead is clearly visible: the larger the read ahead, the closer the transfer rate to the Gigabit Ethernet speed limit (125 MByte/s). However, the performance lags far behind the full wire speed: when r exceeds a certain threshold (48, with eight responders) the throughput drops dramatically as a consequence of network congestion. The switch can sustain a many-to-one pattern as long as the aggregate size of the data collected from the various responders keeps below a certain threshold; a large read ahead parameter violates this condition. The switch under congestion discards a lot of frames. The communication layer of the base DRAID prototype is based on UDP, and does not provide congestion control (Section 2). As a result, the system keeps working but performance drops to very modest levels (not reported in the Figure).

Choosing a small value for the parameter r of read ahead would prevent congestion from occurring, but at the price of a permanently low performance. In addition, the system would not adapt to possible changes in network load. Last but not least, this solution is not scalable because with more and more responders the degree of read ahead should become smaller and smaller. We thus preferred to seek other solutions.

6.2. Second tactic: token passing among responders

An alternate solution is to use a robust (w.r.t. node failures) scheduling algorithm for the responders, so as to avoid

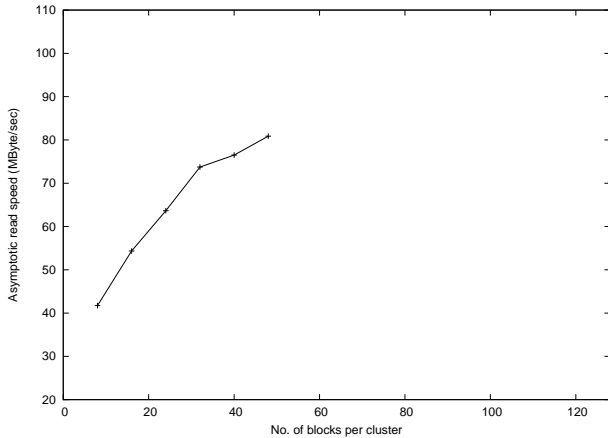


Figure 1. DRAID read speed as a function of the number of blocks per operation, best-effort case.

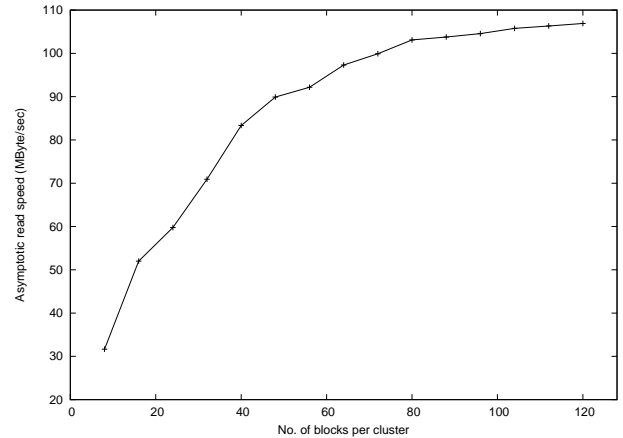


Figure 2. DRAID read speed as a function of the number of blocks per operation, with token-passing congestion prevention.

flooding any single output port of the LAN switch beyond a certain sustainable limit (to be experimentally determined).

For evaluation purposes, we implemented a simplified algorithm by which a static partial ordering is imposed on the set of responders. This is obtained by letting a few “tokens” circulate from one responder to the next one, and delaying transmission of a responder until it gets a token from another responder. At each read request, the first T responders are initially granted a token, so they all can transmit at the same time, whereas the remaining hosts cannot transmit yet. Then, the T responders pass their token to other T responders, which then can transmit, and so on until all responders have done. We have experimentally found that a good balance between serialization among senders and aggregate performance is achieved with $T = 2$.

Figure 2 shows the DRAID asymptotic read throughput as a function of the degree r of read ahead.

With the token-passing tactic for congestion prevention, we were able to play with read ahead parameters far beyond 48, thus approaching a greater transfer rate compared to the best-effort case. However, the token-passing tactic is not such a good idea: the token latency is exposed in between the activities of consecutive responders, with a negative impact on the transfer rate. Indeed the transfer rate only becomes acceptable with unrealistically high degrees of read ahead.

6.3. Third tactic: use TCP instead of UDP

Yet another way to limit congestion might be to use a communication protocol with native congestion control. The communication layer of DRAID is currently based on UDP. TCP could in principle provide a better solution, be-

ing it able to react and self-adapt to congestions.

To investigate this, we made use of the extended “ping pong” microbenchmark described in Section 5 and measured the read throughput as seen by an individual requestor in the cluster. The Nagle feature of Linux TCP was disabled in this experiment. Disks are not involved this time, so the measured performance is expected to over-estimate the real behaviour.

The pretty odd outcome of the experiment is depicted in Figure 3. The interconnect sustains up to two simultaneous TCP flows. With more than two responders, however, the congestion is badly managed. The apparently paradoxical behaviour (less performing with shorter messages, where congestion could be expected to have a smaller impact) can be explained under the hypothesis that the switch near congestion issues pause frames (802.3X) to all senders except one. One of the sender thus continues to transmit during the pause, until end of its own message. Such a strategy can help decreasing the bandwidth contention, but when messages are too short the pause is largely wasted.

6.4. Fourth tactic: responders orchestrated by the requestor

As a further approach, we decided to implement congestion prevention by letting the requestor to decide how many and which responders to ask for stripes at a given time. On a read operation, the requestor P could ask the stripes one subsets at a time rather than all together, thus asking to a subset of a DRAID row at a time rather than asking to all the nodes in the row at the same time. If M is the size of the subset, that is, the number of nodes allowed to respond concurrently, then P can act a congestion control by increasing

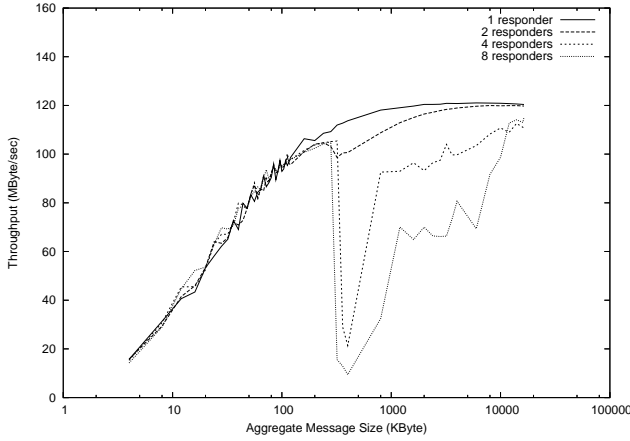


Figure 3. Throughput of many-to-one communication as a function of the total size of received data, with TCP instead of UDP and increasing number of responders.

or decreasing M , in much the same way as TCP acts on the window size. It holds $1 \leq M \leq N + K$, given that each DRAID row counts $N + K$ hosts and at worst one responder at a time can be allowed.

Such a mutable *orchestration* of responders managed by the requestor is more adaptive than the circulation of a fixed number of tokens. Initially, M is set to the allowed maximum of $N + K$. The detection of missing datagrams in a read response causes the requestor host to decrease its local value of M by one; after two consecutive decreases the requestor assumes a real congestion is on the way, so, from then on, M get halved at each further read response with missing datagrams. Conversely, if the requestor sees incoming traffic without missing datagrams, it increases M by one each 5 seconds. In case of no traffic M remains unchanged. We have played experiments in DRAID which shows that, in absence of extra communication traffic, M eventually approaches a level which depends upon the degree of read ahead. A small read ahead decreases the risk of congestion and thus more concurrent responders are allowed (M is near to $N + K$), whereas a greater read ahead increases the congestion hazard and thus less concurrent responders are allowed (M is near to 1). Figure 4 illustrates this behaviour.

Figure 5 shows the asymptotic communication throughput achieved by the many-to-one communication with the orchestration policy. The efficiency seems to improve quite a lot over all the other tactics for congestion prevention, but we must recall that this is just communication performance, without disk accesses.

In the final version we will report read performance including disk access.

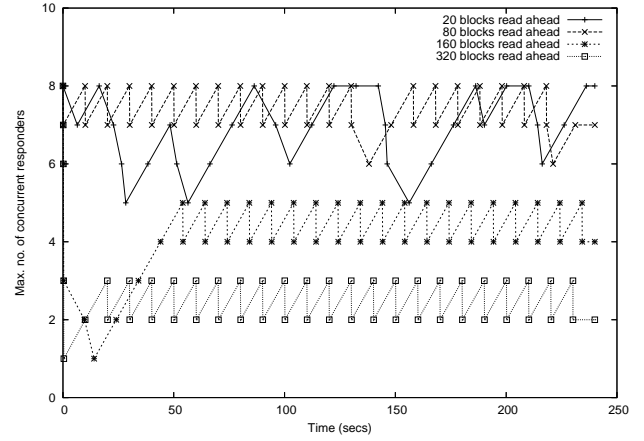


Figure 4. Number of concurrent responders allowed over time by the orchestration policy, with different degrees of read ahead.

However, Figure 5 also shows that a substantial degree of read ahead is necessary in order to attain adequate performance on read operations. This is an effect of the latency inherently incurred by the request-response pattern. To achieve 100 MByte/s we need a read ahead degree of at least 30 (total message size 120 KByte, with 4 KByte blocks), which means each single requested block carries other 29 subsequent blocks; this is expected to increase as disks get involved, because of the additional disk latency. As nothing can be done to optimize over disk latencies, we conclude that the only way to achieve a better efficiency is to tolerate the latency by allowing non-blocking read operations at application level, a feature whose importance has long been recognised for parallel I/O [5].

7. DRAID performance on write

Figure 6 reports the DRAID asymptotic throughput on write, as a function of the degree of write clustering, that is, the number of consecutive 4 KByte blocks aggregated together in a single write operation. A substantial degree of write clustering appears to be a key feature to attain a good write transfer rate.

8. Related work

Despite the great interest around cluster architectures and storage systems, to the best of our knowledge there are quite few documented research efforts on the practical use of clusters as cost-effective, dependable distributed storage systems. Although the idea dates back to the past decade, the performance limits of interconnection networks at that

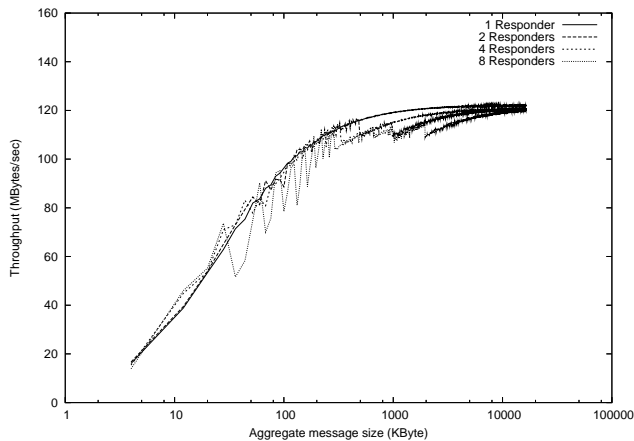


Figure 5. Throughput of many-to-one communication as a function of the total size of received data, with responders orchestrated by the requestor.

time did not encourage much the practical implementations of distributed RAID's.

Swift [4, 10] was one of the first prototypes to practically implement the idea of connecting more independent disks in parallel according to a distributed architecture based on a generic interconnection network, as opposed to the centralized, “hard-wired” architecture proposed for traditional RAID's [12]. The overall organization, however, is client-server rather than peer-to-peer: a distinction is made between client nodes, hosting the application, and server nodes, hosting the storage resources.

The Petal experiment [9] proposed a distributed storage system consisting of networked storage servers. The project was the first in implementing the concept of a distributed RAID. A Petal system could only tolerate single failures, and provide a single end-point for service access, using a custom RPC-style interface. Petal implemented the global address space of the virtual block device at the user level, rather than in the OS kernel; this choice made it necessary to develop a special-purpose file system running at user level as well. A distributed global lock functionality was provided, in order to support a distributed file system called Frangipani [20]. The Swarm Scalable Storage System [6], based on the GNU/Linux OS, shares many of its basic principles and features with Petal, but provides no data redundancy or distributed lock facilities.

Tertiary disk [3] is a reliable storage system built with a cluster of PCs. Each single node in the cluster comprises two PCs, each with its own SCSI controller, and as many as 14 SCSI disks shared by both PCs. It applies the RAID-5 pattern of data distribution across disks, thus tolerating at most one single disk failure. Duplication of hardware

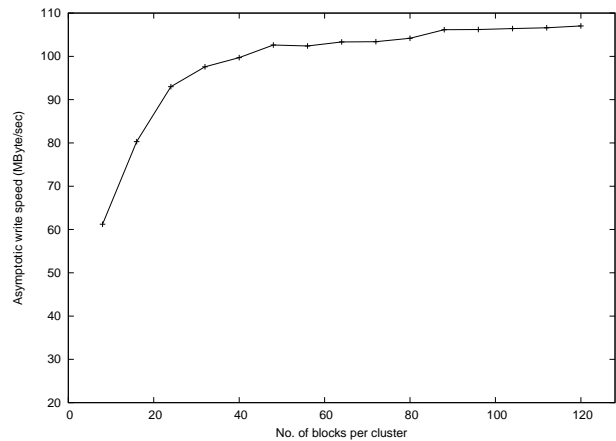


Figure 6. DRAID write speed as a function of the number of blocks per operation.

resources (PCs and SCSI controllers) at the node level is meant to improve robustness. A log-based serverless network file system called xFS [1] runs on top of Tertiary Disk.

The RAID-x experiment [8] is the most recently documented prototype of a distributed RAID we are aware of, and also the first one to implement a SIOS virtual block device. Nevertheless, the main focus of RAID-x seems on the proposal for a RAID mode called *orthogonal striping and mirroring*. With orthogonal striping and mirroring, a RAID tolerates at most one disk failure regardless of the total number of workstations involved in the distributed RAID. Tolerating at most one failure clearly does not scale up with the number of nodes, especially when the nodes are workstations operated by humans and as such exposed to any kind of physical abuse. As far as we know, DRAID is the only existing distributed RAID to both support a SIOS and tolerate multiple failures.

A slightly different idea, proposed in [21], is to use remote disks to store parity information of the local disk. This way, in a large cluster of PCs with a disk on each node, each disk becomes reliable from the standpoint of the owner node at the price of giving away some local disk space to store the redundancy from some other node. There is no SIOS, but after all the focus of that proposal is on making each individual disk reliable, rather than aggregating disks.

It must also be said that the distributed RAID architecture is just one possible approach to distributed storage; we could mention at least other two such ones:

- parallel file systems, like PVFS [5] and the GPFS featured by IBM, in which the focus is more on parallel accesses to possibly distributed files;
- distributed file systems, like NFS [16], Coda [18], Intermezzo [11], and the GFS featured by Sistina Soft-

ware, which either hide the physical distribution of files across several remote disks, or just aggregate a collection of individual remote file systems under a single, global and possibly shared file system abstraction.

In all the above cases, the SIOS abstraction is achieved at the level of file system rather than block device. This has the drawback of being committed to a specific file system; conversely, a distributed block device can support any existing file system, with small modifications to support or forbid concurrent writes on shared data structures.

9. Conclusions and open points

The encouraging results shown in this paper have been achieved without any specialized piece of hardware or driver, and without even modifying the existing OS kernel. All the software is self-contained into a single Linux kernel module.

DRAID offers a SIOS abstraction, and is able to tolerate a site-configurable number of disk failures. No other distributed RAID is currently able to provide both such features; yet, SIOS and multiple fault tolerance are of decisive importance for so-called “I/O-centric clusters”, namely, clusters of PCs aimed at offering storage services and/or running I/O-intensive tasks. DRAID can be expanded by simple addition of new PCs with local disks, up to saturation of the LAN bandwidth. Larger configurations can be achieved by increasing the throughput of the interconnect; this can be obtained by leveraging next generation LAN hardware.

The potential of the SIOS block device abstraction for parallel I/O deserves further investigation and evaluation. Another interesting point for investigation is on the effectiveness of DRAID as a support for a shared file system. A promising approach we are going to explore is based on modifying the Linux Virtual File System (VFS) layer, namely, the kernel-level stub to the many Linux file systems, in order to grab information about concurrent accesses to files; this information can be used to minimize the overhead of a distributed algorithm for buffer cache coherency that DRAID still lacks.

References

- [1] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless Network File Systems. In *Proc. of the 15th ACM Symp. on Operating Systems Principles (SOSP)*. *ACM Operating Systems Review*, 29(5), 1995.
- [2] S. Asami, N. Talagala, T. Anderson, K. Lutz, and D. Patterson. The Design of Large-Scale, Do-It-Yourself RAIDs. <http://www.cs.berkeley.edu/pattrsn/papers.html>, Nov. 1995.
- [3] S. Asami, N. Talagala, and D. Patterson. Designing a Self-Maintaining Storage System. In *16th IEEE Symp. on Mass Storage Systems*, San Diego, CA, Mar. 1999. IEEE.
- [4] L. F. Cabrera and D. D. E. Long. Swift: a storage architecture for large objects. In *11th IEEE Symp. on Mass Storage Systems*, pages 123–128. IEEE, Oct. 1991.
- [5] P. H. Carns, W. B. L. III, R. B. Ross, and R. Thakur. PVFS: A Parallel File System For Linux Clusters. In *Proc. of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, Oct. 2000.
- [6] J. H. Hartman, I. Murdock, and T. Spalink. The Swarm Scalable Storage System. In *Proc. of the 19th Int’l Conf. on Distributed Computing Systems (ICDCS)*, pages 74–81, Austin, TX, May 1999. IEEE.
- [7] K. Hwang, H. Jin, E. Chow, C. Wang, and Z. Xu. Designing SSI Clusters with Hierarchical Checkpointing and Single I/O Space. *IEEE Concurrency*, 7(1):60–69, 1999.
- [8] R. S. C. H. Kai Hwang, Hai Jin. Orthogonal Striping and Mirroring in Distributed RAID for I/O-Centric Cluster Computing. *IEEE Trans. on Parallel and Distributed Systems*, 13(1):26–44, 2002.
- [9] E. K. Lee and C. A. Thekkath. Petal: Distributed Virtual Disks. In *Proc. of the 7th Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, Cambridge, MA, Oct. 1996. ACM Press.
- [10] D. D. E. Long and B. R. Montague. Swift/RAID: A Distributed RAID System. *Computing Systems*, 7(3):333–359, 1994.
- [11] P. J. Braam, et al. Intermezzo File System Home, <http://www.inter-mezzo.org/>, 2001.
- [12] D. Patterson, G. Gibson, and R. H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of ACM SIGMOD Annual Conference*, pages 109–116, Chicago, IL, June 1988. ACM Press.
- [13] W. W. Peterson and J. E. J. Weldon. *Error-Correcting Codes, 2nd ed.* MIT Press, Cambridge, MA, 1972.
- [14] G. F. Pfi ster. The Varieties of Single System Image. In *Proc. of IEEE Workshop on Advances in Parallel and Distributed Systems*, pages 59–63. IEEE, 1993.
- [15] A. L. N. Reddy and P. Banerjee. An Evaluation of Multiple-Disk I/O Systems. *IEEE Trans. on Computers*, 38(12):1680–1690, Dec. 1989.
- [16] S. Shepler, et al. RFC 3010: NFS version 4 protocol, <http://www.ietf.org/rfc/rfc3010.txt>, Dec. 2000.
- [17] K. Salem and H. G. Molina. Disk Striping. In *Proceedings of the Second International Conference on Data Engineering*, pages 336–342, Los Angeles, CA, Feb. 1986. IEEE Computer Society.
- [18] M. Satyanarayanan, J. J. Kistler, and E. H. Siegel. Coda: A Resilient Distributed File System. In *IEEE Workshop on Workstation Operating Systems*, Cambridge, MA, Nov. 1987.
- [19] M. Stonebraker and G. A. Schloss. Distributed RAID - A New Multiple Copy Algorithm. In *Proc. of the 6th Int’l Conf. on Data Engineering*, pages 430–437, Los Angeles, CA, Feb. 1990. IEEE.
- [20] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A Scalable Distributed File System. In *Proc. of the 16th ACM Symp. on Operating Systems Principles (SOSP)*, Saint Malo, France, Oct. 1997. ACM Press.

- [21] A. Wiebalck, P. T. Breuer, V. Lindenstruth, and T. M. Steinbeck. Fault-tolerant distributed mass storage for lhc computing. In *Proceedings of the Third IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 266–275, Tokyo, Japan, May 2003. IEEE Computer Society Press.