Efficient Many-to-one Communication for a Distributed RAID *

Alessandro Di Marco, Giuseppe Ciaccio

DISI, Università di Genova
via Dodecaneso, 35
16146 Genova, Italy
E-mail: {dmr,ciaccio}@disi.unige.it

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 about issues of communication performance in 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 inefficiency of communication when a client reads data stripes from a number of remote servers in a Gigabit Ethernet LAN. The congestion caused by such many-to-one communication pattern has been faced in multiple ways, but the best result has been obtained by modifying the traditional, and unsuccessful, congestion avoidance policy of TCP/IP.*

Keywords: RAID, network-attached storage, many-to-one communication, congestion avoidance, 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 scientific 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 [16, 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 [13].

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* [18] can provide a capable, dependable, and cost-effective storage system made out of all the unused and inexpensive hard disks provided by the partecipants. In a distributed RAID, all individual disk requests are managed by a proper distributed software running on all participant hosts; this software manages data

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

striping and replication over the disks provided by the participants, according to some predetermined mapping, using the existing networking infrastructure for transmitting 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 [18] 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) [6] which is a special case of a more general feature called *single system image* [14]. 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 equally 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 challenges, namely, multiple concurrent writes to shared blocks, coherency among the local block caches of each participant host, and tolerance to transient as well as persistent network failures.

But all distributed RAIDs also suffer a commonly overlooked illness, that is, bad performance of read operations. Each data block of the storage is striped across a number of remote servers. Thus, a *requestor* wishing to read one or more data blocks has to send requests to a number of remote hosts, called the *responders*; shortly after, as a consequence, the client gets a burst of data packets coming from those responders. Such a *many-to-one* communication pattern usually causes a network congestion near the client, with a huge degradation of performance. The congestion problem is exhacerbated by other two typical drawbacks of distributed environments, namely:

- the lack of coordination among participants, and the lack of scalability of whatever coordination algorithm for the responders;
- the need to mask the network latency, which forces each requestor to read more consecutive data blocks than immediately needed (the so called "read-ahead" optimization); this increases the amount of data conveyed towards the congestion point.

Similar communication problems arise in case of concurrent writes to striped data. In this case, congestions can occur because of multiple clients attempting to update data onto the same set of remote servers. But this only occurs in case of concurrent writes to shared blocks, as opposed to what happens with read operations, which always lead to congestion regardless of whether they are concurrent or not. So, a cure for read congestion is crucial to the overall system performance.

In this paper we report about DRAID, a distributed RAID device in which the challenges posed by concurrent writes and network fault tolerance were successfully faced long ago [11], whereas the efficiency of read operations has only recently attained satisfactory levels thanks to the replacement of the congestion avoidance policy of TCP/IP by a more effective one.

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

DRAID is a virtual block device (not a filesystem) that distributes blocks across a set of networked workstations, by first computing a Reed-Solomon redundant encoding of each block, then striping the resulting data among disks (or any other block device) made available by the participant hosts.

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*, $B_1 B_2 ... B_N$, which is then extended with additional K redundant segments $P_1 P_2 ... P_K$ computed by the Reed-Solomon encoder. All the Bs and the Ps segments have same size. The resulting *data stripe*, $B_1 B_2 ... B_N P_1 P_2 ... P_K$, 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 $B_1B_2...B_N$. 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.

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. 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.

DRAID also implements suitable policies for handling node failures and concurrent writes. Due to space constraints we could not report here about these issues; the interested readers may resort to [11].

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. 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 the aforementioned *read ahead*. Each read operation to a given virtual block B_l actually triggers reading a whole sequence $B_l, B_{l+1}, B_{l+2}, ..., B_{l+r-1}$ 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.

3. How to improve DRAID performance on read

In this Section we discuss performance of the bare DRAID virtual block device, with no file system on top of it, on read operations. Performance of write operations did not pose any particular challenge [11]. Conversely, the congestion problem caused by read operations required far more investigation in order to find an effective solution.

The measurement testbed is a cluster of dual-CPU PCs, each with two Xeon 2.8 GHz processors, a Maxtor Diamond Max Plus 9 80 GByte ATA disk, and an Intel PRO/1000 Gigabit Ethernet adapter. The PCs are networked by an Extreme Networks Summit 7i Gigabit Ethernet switch, with copper wiring. The switch supports IEEE 802.3X flow control, but never generates pause frames on congestion; this means, the switch is unable to prevent congestion from occurring, and the entire congestion avoidance burden is on the communication protocol.

The entire cluster is arranged into DRAID 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.

The block size of the DRAID virtual block device was set to the maximum allowed by Linux 2.6, namely, the memory page size (4 KByte with the IA32 processor architecture). To improve performance, the MTU size of the network was such that a block of data could always be arranged into a single network packet.

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.

As already pointed out, in order to read any given DRAID virtual block, a given requestor P issues $N + K$ read requests to as many responders in a short time interval. Shortly after, as a consequence, P gets a burst of packets coming from those responders. Such a many-to-one communication pattern causes a network congestion. In search of solutions to such congestion problem we have played with different tactics. After discarding obviously ineffective solutions (reducing the amount of read ahead, and coordinating the responders with circulating tokens), we came up with the two tactics accounted below.

3.1. First tactic: responders orchestrated by the requestor

As a first 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 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.

The communication layer used in the DRAID prototype for this experiment was UDP, augmented with datagram retransmission but lacking congestion control.

The 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 1 shows the asymptotic communication throughput achieved by the many-to-one communication with the orchestration policy. This measurement has been accomplished with the aid of the extended ping-pong microbenchmark described in Section 3. Disks were not involved this time, so the measured performance is expected to overestimate the real behaviour.

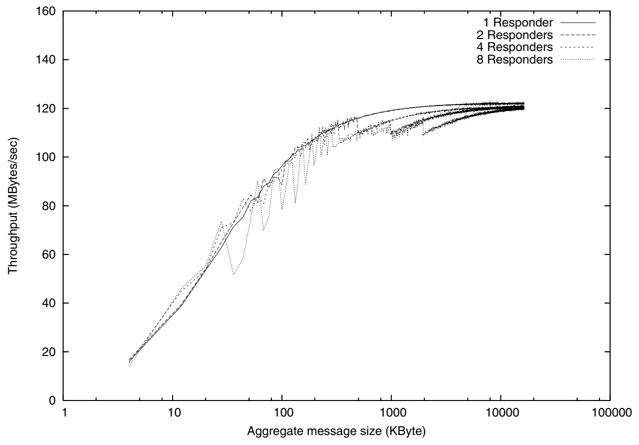

Figure 1. Throughput of many-to-one communication as a function of the total size of received data, with responders orchestrated by the requestor.

With this approach to congestion prevention, the throughput of DRAID read operations might indeed approach the Gigabit Ethernet limit. However, it is also apparent that a substantial degree of read ahead is needed in order to attain adequate performance. 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 (aggregate 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 [3].

3.2. Second tactic: use TCP instead of UDP

Another way to limit the network congestion is to use a communication protocol with native congestion control. TCP is commonly believed to be able to react and self-adapt to congestions, so it was a promising alternative for our

DRAID. We therefore replaced UDP by TCP in the DRAID prototype.

To evaluate performance, we made again use of the extended ping-pong benchmark described in Section 3 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 were not involved, so the measured performance is expected to over-estimate the real behaviour. The number of responders ranged from 1 to 8. Aggregate message sizes were constrained to be multiple of the block size. For each message size, the microbenchmark runs repeated request-response cycles for two seconds, to allow the network to stabilize, then continues for two more seconds and counts how many request-response cycles are completed in this latter interval. The average throughput of a request-response cycle is thus easily computed as a function of the aggregate message size.

The odd outcome of the experiment is depicted in Figure 2. As apparent from the Figure, the interconnect seems to sustain up to two simultaneous TCP/IP flows. With more than two responders, however, the congestion is badly managed, and a huge and sudden performance drop appears at around 300 KByte (the more the responders, the smaller this threshold).

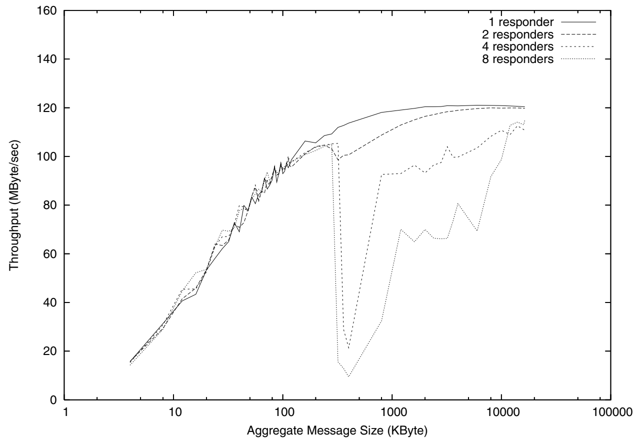

Figure 2. Throughput of many-to-one communication as a function of the aggregate size of received data, using Linux TCP/IP with increasing number of responders.

Actually, Linux provides many flavours of TCP/IP: Reno, BIC, Vegas, and Westwood. TCP BIC is the current default, but, as we can see in Figure 3, the other flavours do not work better. In all cases, the selective acknowledge (SACK) feature was also enabled.

We decided to focus on TCP Reno, which uses the simpler and most "ancient" congestion avoidance policy, proposed by Jacobson [7], and analyzed the behaviour of the congestion windows during a many-to-one communication test. We properly instrumented the Linux TCP kernel code, then ran a six-to-one communication test with message sizes falling in the "bad behaviour" region highlighted in Figure 2. The results reported in Figure 4 are from a test with aggregate message size of 327680 bytes (80 blocks). The periodical and synchronous downsizings of all windows mark the occurrences of as many network congestion episodes.

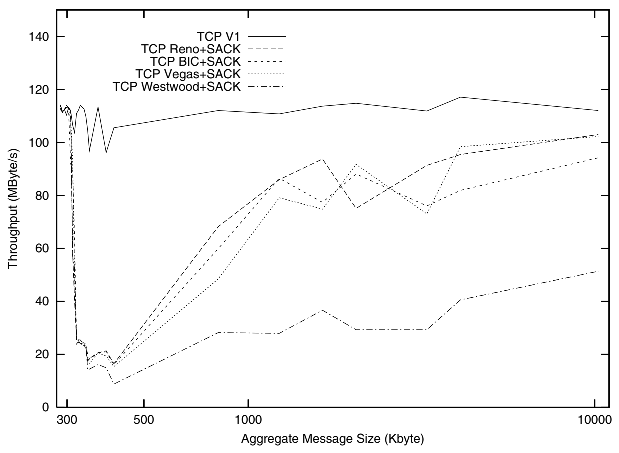

Figure 3. Throughput of many-to-one communication as a function of the aggregate size of received data, with six responders and various flavours of Linux TCP/IP. The curve from TCP/IP with a modified congestion avoidance policy (TCP V1) shows the improvement achieved by adopting a different congestion avoidance policy.

By inspecting the curves it seems that, at least in principle, there are two reasons for poor performance of TCP under the many-to-one congestion pattern: on one hand, congestion windows are too drastically downsized at each congestion episode, with an excessive decrease of network utilization; on the other hand, congestion windows are too early allowed to grow past the limit encountered at the previous congestion episode, with a new congestion episode as a consequence. Better TCP performance could thus be achieved by using different policies of congestion avoidance.

In the TCP jargon, the congestion avoidance policy is described in terms of two variables, namely, the congestion window size $CWND$, and the slow start threshold $SSTH$. In the Reno congestion avoidance scheme, $CWND$ grows fast until $SSTH$ is reached, then continues to grow past $SSTH$ at an exponentially slower and slower pace; when a congestion is detected, $SSTH$ takes half the current value

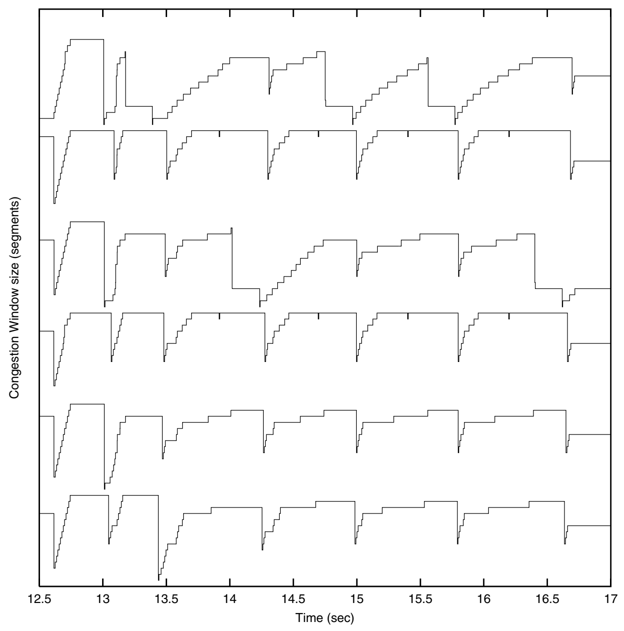

Figure 4. Size of congestion windows when six TCP Reno flows converge into a six-to-one communication lasting 4 seconds. The size of each window ranges between a minimum of 1 and a maximum of 15 segments. Aggregate message size is 327680 bytes (80 blocks). The repeated congestions are marked by the drops of window sizes.

of $CWND$ and the growth of $CWND$ restarts from 1. In search for better TCP performance we have experimented with the following five different variants of the above congestion avoidance policy:

- V1: $CWND$ is never allowed to grow past $SSTH$; when $CWND$ reaches $SSTH$, both are allowed to increase by 1 every 1000 acknowledged segments if no congestion arises meanwhile. On congestion, V1 behaves like Reno.
- V2: like V1, but $SSTH$ is set to half its own current value in case of congestion, rather than half the current $CWND$ (if $SSTH$ happens to be yet undefined when congestion arises, as it is the case when the connection has started very recently, $SSTH$ is set to $CWND - 1$).
- V3: like V2, but $CWND$ takes half its current value in case of congestion, rather than (too drastically?) restarting from 1.
- V4: like V3, but $SSTH$ is just decreased by 1 in case of congestion, rather than halved.
- V5: like V4, but also $CWND$ is just decreased by 1 in case of congestion, rather than halved.

All the variants share a common feature, namely, $CWND$ grows much slower than in TCP Reno once $SSTH$ is reached. The differences are in how they behave in case of congestion, with less and less drastic behaviours from V1 to V5.

All the five different variants of TCP, from V1 to V5, show a substantial improvement over TCP Reno (as well as the other established TCP flavours) when engaged in a many-to-one congestion pattern. This is clearly displayed by Figure 5. The huge performance gap affecting traditional TCP is almost completely filled by variant V1 (Figure 3). Thus, it seems that a severe limitation on the growth of congestion windows past the slow start threshold during congestion-free phases is the key ingredient for good many-to-one communication throughput with TCP. However, less drastic policies in case of congestion detection appear to be slightly detrimental. A less drastic window downsizing improves network utilization but also increases the probability of another congestion episode in the short term; these measurements apparently account for the latter aspect prevailing over the former one.

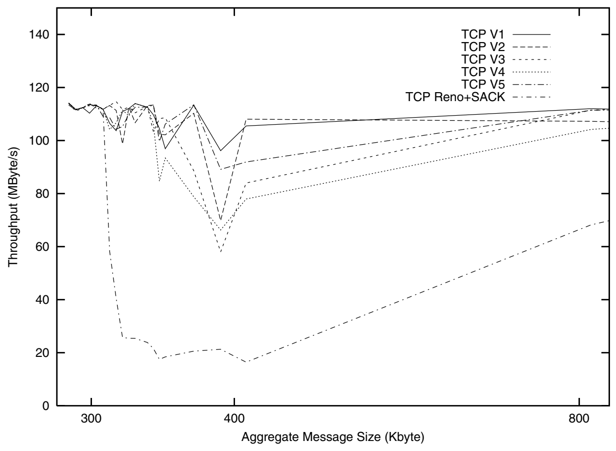

Figure 5. Throughput of TCP/IP many-to-one communication as a function of the total size of received data, with six responders and five non-standard congestion avoidance policies. The curve from TCP Reno is shown as a reference. TCP V1 seems to perform best. In any case, the improvement over traditional congestion avoidance policies is sharply evident.

Overall, the modifications introduced into the TCP congestion avoidance mechanism make it possible for the DRAID read operations to approach the Gigabit Ethernet throughput limits, in much the same way as the other approach based on UDP (Section 3.1), with similar considerations concerning the substantial degree of read ahead needed to attain adequate performance.

4. Related work

Swift [2, 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 RAIDs. 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 [19]. The Swarm Scalable Storage System [5], 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 [1] 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 resources (PCs and SCSI controllers) at the node level is meant to improve robustness.

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 [20], 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 [3] and the GPFS featured by IBM, in which the focus is more on parallel accesses to possibly distributed files;
- distributed file systems, like Coda [17], Intermezzo [12], Lustre [4], and the GFS featured by Sistina Software, 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.

5. Conclusions and open points

In the end, we have found two successful ways to counter the many-to-one network congestions caused by read operations of our DRAID prototype. The first of them required the responders to be properly orchestrated by a UDP-speaking requestor. The latter acted upon the congestion avoidance policy of TCP. We believe the latter approach wins, because it allows a more transparent interaction between requestor and responders, which turns into a much simpler implementation of the communication sub-layer of DRAID.

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 currently exploring 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] S. Asami, N. Talagala, and D. Patterson. Designing a Self-Maintaining Storage System. In *16th IEEE Symp. on Mass Storage Systems*, San Diego, CA, March 1999. IEEE.

[2] 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, October 1991.

[3] P. H. Carns, W. B. Ligon 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, October 2000.

[4] Cluster File Systems, Inc. Lustre: scalable, secure, robust, highly-available cluster file system, http://www.lustre.org.

[5] 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.

[6] 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.

[7] V. Jacobson. Congestion Avoidance and Control. In *Proc. of ACM SIGCOMM*, Stanford, CA, August 1988. ACM.

[8] Roy S. C. Ho 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, October 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] A. Di Marco, G. Chiola, and G. Ciaccio. Using a Gigabit Ethernet Cluster as a Distributed Disk Array with Multiple Fault Tolerance. In *Proc. of the 28th IEEE conf. on Local Computer Networks (LCN 2003), workshop on High-Speed Local Networks (HSLN)*, Bonn-Königswinter, Germany, October 2003. IEEE.

[12] P. J. Braam, et al. Intermezzo File System Home, http://www.inter-mezzo.org/, 2001.

[13] 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.

[14] G. F. Pfister. 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, December 1989.

[16] K. Salem and H. Garcia Molina. Disk Striping. In *Proceedings of the Second International Conference on Data Engineering*, pages 336–342, Los Angeles, CA, February 1986. IEEE Computer Society.

[17] M. Satyanarayanan, J. J. Kistler, and E. H. Siegel. Coda: A Resilient Distributed File System. In *IEEE Workshop on Workstation Operating Systems*, Cambridge, MA, November 1987.

[18] 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, February 1990. IEEE.

[19] 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, October 1997. ACM Press.

[20] A. Wiebalck, P. T. Breuer, V. Lindenstruth, and T. M. Steinbeck. Fault-Tolerant Distributed Mass Storage for LHC Computing. In *Proc. of CCGRID*, Tokyo, Japan, May 2003. IEEE.