

# Improving I/O Performance of Clustered Storage Systems by Adaptive Request Distribution

Changxun Wu\* and Randal Burns  
Department of Computer Science  
Johns Hopkins University  
{wu,randal}@cs.jhu.edu

## Abstract

*We develop an adaptive load distribution protocol for logical volume I/O workload in clustered storage systems. It exploits data redundancy among decentralized storage servers to dynamically route I/O workload on a per-request basis, offering short-term load balancing and improved I/O performance. Our protocol builds on tunable hashing techniques and is based purely on client logic. Therefore, it does not limit system scalability and requires no change to the existing infrastructure. It distributes the I/O requests of a client to storage servers selected adaptively by a decentralized tunable hashing scheme, and, applies different policies to read and write requests. It also makes no assumption about inter-server communication latency and thus is robust to different network configurations. It supports both replication and erasure coding data redundancy schemes. Experimental results show that our protocol performs closely to a centralized load-balancing algorithm and verify the robustness of our protocol.*

## 1 Introduction

Many research institutions and enterprises now face the challenge of storing and managing a large amount of data. Even worse, the data are usually generated and stored in a decentralized fashion. Such “islands of storage” make it inconvenient for applications to locate data, endanger data availability, and create data management challenges.

To address this problem, a recent trend builds clustered storage systems on top of heterogeneous, autonomous storage sites [2, 4, 7, 12, 13, 16, 22, 25]. Such storage consolidation and resource aggregation facilitate building scalable, low-cost storage systems able to handle large amounts of data and workload that would overburden any single

storage site. With logical volumes constructed across the sites, a distributed storage system appears as a single centralized storage pool with a standard block interface, greatly simplifying data management.

However, such distributed storage systems have to carefully deal with the issues of data reliability and load distribution, which could severely affect system performance. Their distributed hardware infrastructure is vulnerable to component failures and network partitions, due to the wide use of commodity hardware and the use of a general-purpose network as the primary, system-level interconnect. Therefore, data redundancy techniques, such as erasure coding or replication [14, 27], are needed to provide data reliability guarantees comparable to centralized, enterprise-class storage systems. Meanwhile, their distributed infrastructure makes it difficult to collect global knowledge in order to efficiently distribute load. It also makes centralized load balancing techniques inappropriate: a potential single point of failure and performance bottleneck. Furthermore, the heterogeneity among storage server software and networks jeopardizes the interoperability among sites.

Load balancing strategies in storage subsystems tend to be either simple or long-term, because it is costly to move persistently stored data once they are placed. The simple strategies either stripe data and assume that applications will access entire stripes or place and access data in some simple randomized fashions, such as simple hashing. However, striping across heterogeneous disks could lead to load imbalance and it has been shown that simple randomization fails to handle heterogeneity [29]. The long-term approach moves data to rebalance systems in response to long-term load trends, such as Hippodrome [3]. Long-term load balancing happens infrequently and is not very responsive to short-term load fluctuations. Moreover, it fails to provide fine-grain load distribution capabilities, *e.g.* on a per request basis. So, the gap here is in providing short-term load balancing (on an individual request basis) among storage sites.

\* C. Wu is currently at Google.

We construct a load distribution protocol that balances short-term load fluctuations in I/O requests to logical volumes built from heterogeneous storage servers. Our protocol is based on a tunable hashing scheme derived from ANU randomization [30]. The tunable hashing scheme dynamically routes a client's I/O requests to appropriate servers on a per-request basis and adaptively tunes itself based on client-observed server performances. It also applies different policies to read and write I/O requests. Our protocol is applicable to distributed logical volumes using either replication or erasure coding techniques, such as Information Dispersal [23], Reed-Solomon coding [28], and fountain codes [21].

Our research objective can be framed in comparison to D-SPTF [20], which performs short-term load balancing for read requests through device scheduling and assumes each data item is replicated on multiple servers in a clustered storage system. D-SPTF initiates I/O requests at all replicas. The first replica to serve the I/O notifies the other replicas of the estimated completion time of the request. The other replicas cancel the outstanding request. The "late binding" of a request to a disk allows the system to achieve minimum response time. It also has the effect of realizing load balance, because jobs are canceled from slower (more loaded) servers. Thus, the queues at slower servers are shortened. For similar reasons, it handles performance heterogeneity among servers implicitly; less powerful servers process fewer jobs because they take longer to complete each task on average. D-SPTF has proven to be successful. Its drawbacks include

- 1) a tightly coupled architecture: all nodes must run the D-SPTF protocol that allows for node-to-node communication for job interruption;
- 2) messaging overhead: D-SPTF sends multiple network messages for each I/O, which is only suitable for clusters built on high-performance interconnects.

In contrast to D-SPTF, our architecture is designed to support multiple types of heterogeneity. In addition to performance heterogeneity, this includes operating system and hardware heterogeneity. The intent is to permit distributed logical volumes to be constructed using parts from different manufactures that support industry standard network I/O interfaces, such as iSCSI [24]. As a corollary, all technology components are isolated within the storage client (initiator). Our system requires no changes to storage server nodes (targets) and network storage protocols. Each storage server node is autonomous.

Our architecture also supports the construction of distributed logical volumes across a wide-variety of network configurations. D-SPTF aggressively exchanges messages among server nodes to select a serving node for each individual request. Thus, it requires low-latency node-to-node network communication and is only suitable for

computing clusters and brick-based architectures. Instead, in our system, nodes do not exchange messages among each other, *i.e.*, our system does not use node-to-node communication to realize load balance. Therefore, our system makes no assumptions of inter-server communication latency and is robust to different network configurations. It is able to incorporate storage servers that reside on multiple networks with different configurations.

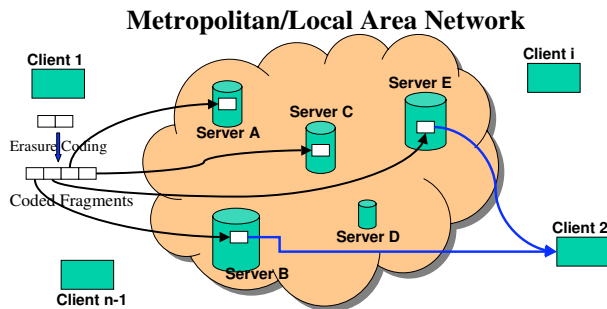
## 2 Related Work

There have been considerable research efforts to build large virtual disks from a collection of decentralized smaller components, such as TickerTAIP [5] and Petal [17]. More recently, brick-based storage systems have been proposed as an alternative to centralized enterprise disk arrays [6, 9, 18]. These efforts often describe a tightly coupled architecture with each participating storage server running the same software module. Such modules are complicated and could be non-open source, resulting in vendor lock-in – existing systems could only be upgraded with components from the same vendor. Moreover, TickerTAIP and brick-based storage systems require high-performance, reliable interconnects.

Another research trend builds distributed storage systems over wide-area networks [7, 16, 25, 31, 32]. For example, OceanStore is a global persistent data store providing a consistent, highly-available, and durable storage utility atop an infrastructure comprised of untrusted servers [16]. When accessing data in such systems, clients often explicitly favor servers that are close due to network cost. In contrast, we target systems within a short network distance, such that load balance dominates network latency for client performance. In other words, clients favor less loaded servers rather than nearby servers.

Most existing load balancing techniques do not provide efficient short-term load balancing support for distributed logical volumes built from heterogeneous resources. Striping [10] and simple hashing are long-standing techniques to achieve load balance in data systems. However, they are not suitable for heterogeneous environments. Some dynamic techniques [3, 4] tune data placement to handle long-term imbalance. While such techniques are not sensitive to short-term load fluctuations, they efficiently deal with long-term load trends and would be beneficial for our target environments as well. Instead of obviating the need for data movement to handle long-term imbalance, short-term load balancing compliments such techniques to further improve system performance.

D-SPTF [20] most closely resembles our work in spirit. It also exploits data redundancy to provide short-term load balancing among participating storage nodes. However, D-SPTF load balances read requests only. It performs the same write operation on all replicas indiscriminately –



**Figure 1. Clustered storage system architecture.**

writing to local nonvolatile RAM memory (NVRAM) and flushing to disk later. Our protocol instead allows different write operations among the nodes and balances both read and write requests. Also, D-SPTF is suitable for tightly coupled systems only and requires each participating server node to install new software implementing the D-SPTF protocol. Our protocol works in a wide range of network environments and does not require such software updates on server nodes. Furthermore, while D-SPTF works well with replicated data, our results show that it does not perform as well with erasure-coded data. The high messaging cost of D-SPTF makes it relatively inefficient for selecting multiple storage nodes (instead of a single replica) in erasure-coding based systems.

Some industry vendors, such as Acopia [1], provide solutions that load balance distributed file servers. Acopia puts a proprietary appliance at each storage site to dynamically distribute application workload across multiple sites in real-time to balance load. Our work is different from Acopia in that our solution is based purely on client logic. Thus, there is no extra hardware installation needed and no changes to the existing network infrastructure.

### 3 System Architecture

We build our load distribution protocol for I/O requests to distributed logical volumes built in clustered storage systems. We assume the logical volumes employ either replication or erasure coding data redundancy schemes to realize data reliability. A brief review of the two data redundancy schemes and our target architecture helps the discussion.

Replication and erasure coding are the most widely deployed data redundancy techniques to address reliability issues in distributed storage systems. Replication techniques maintain multiple identical copies of the same data item at separate physical locations to survive disk failures and network partitions. Erasure coding computes checksum information from the original data and divides the coded

data into multiple blocks to be distributed across decentralized storage components. More specifically, erasure-coding schemes work as following: it divides the original data item into  $m$  equal-size data blocks and encodes them into  $n$  equal-size coded data fragments where  $n > m$ . Then any  $m$  of the  $n$  coded data fragments can be used to restore the original data item. We call such erasure codes  $m$ -of- $n$  codes. Erasure-coding has been shown to be more space efficient than replication and provides comparable availability guarantees [27]. Therefore, it has been adopted in many recent networked storage systems [9, 11, 13, 22]. Both replication and erasure-coding provide data reliability at the expense of increased storage overhead.

Figure 1 illustrates the architecture of a prototypical clustered storage system built from heterogeneous and autonomous storage servers. Servers are heterogeneous; each server has different performance characteristics, hardware, software, and capacity. Servers are also autonomous in that each server operates in isolation without communication or shared-state with other storage servers. We treat each participating storage server as a single node. We assume that the nodes are physically isolated and connected by a general-purpose IP network. Each node supports industry standard network I/O interfaces such as iSCSI [24]. In our target environment, server nodes are geographically close to each other, for example in the same company or university. In other words, clients observe nearly uniform network latency to all server nodes and load balance dominates network distance for I/O performance.

Logical volumes are constructed across the nodes to present a simple block interface to applications running on clients. This virtualization technique maintains logical volume metadata, handles the initial logical volume data placement, and provides logical-to-physical address translations. In our prototype, we implemented a virtualization technique called V:Drive [4], which is out-of-band and sensitive to both heterogeneous capacities and long-term load imbalance<sup>1</sup>. However, our load distribution protocol works with any generic storage virtualization solution. The prototypical clustered storage system supports both replication and erasure coding to achieve data reliability. Administrators specify a data redundancy policy for each newly created logical volume. For example, 3-way replication or 2-of-4 Luby Transform code [19].

An important parameter for each constructed logical volume is the size of its data placement unit or, in other words, its segment size. A logical volume is divided into multiple, equal-size segments. Each replica or erasure-coded fragment of a segment is allocated continuously on

<sup>1</sup>It runs on a dedicated machine and requires no replicated state among clients. Alternatively, it could be realized as a software module on the clients and requires a few global parameters to be synchronized periodically.

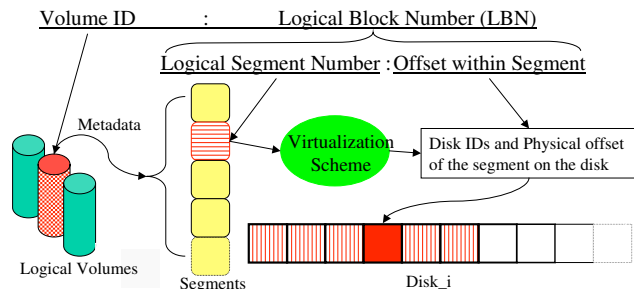


Figure 2. Logical address translation.

a server node. While larger segment size reduces the size of storage virtualization metadata and requires fewer I/O requests to access sequential data, smaller segment size provides better long-term load balancing as well as higher throughput when accessing data. We leave the choice of segment size to the user for customized application support. Such issues are orthogonal to our short-term load balancing goals.

In a typical access, an application running on a client asks for some specific data block in a logical volume. The client first sends the logical volume id and the logical volume block number (LBN) to the virtualization scheme used. Using the segment size (SS) of the logical volume, the logical segment number (LSN) could be easily calculated as  $LSN = LBN/SS$ . The virtualization scheme then uses the concatenation of the logical volume id, the LSN, and some other information, e.g. replica id or erasure-coded fragment id, to identify the set of nodes that store a copy or a coded fragment of the segment where the requested data block resides. The virtualization scheme also computes the physical addresses of the requested block on those nodes for the client. After receiving the response from the virtualization scheme, the client uses the adaptive resource selection (ARS) algorithm described in Section 4 to choose one or more nodes (out of the node set returned by virtualization), and, then, sends an I/O request to each chosen node. Figure 2 shows the logical address translation procedure.

Figure 1 also illustrates another usage scenario. Client 1 produces a new block of data, erasure-codes it with a 2-of-4 erasure code scheme, and then stores the erasure-coded data on multiple storage nodes (A, B, C, and E). To decide which nodes to store the erasure-coded fragments, client 1 queries the virtualization scheme by providing the logical volume id and the LBN of the new data block. Using the same logical volume id and LBN, client 2 identifies the same set of nodes (A, B, C, and E) from the virtualization scheme. It then employs the ARS algorithm to adaptively pick minimum necessary number of nodes (Site B and E)

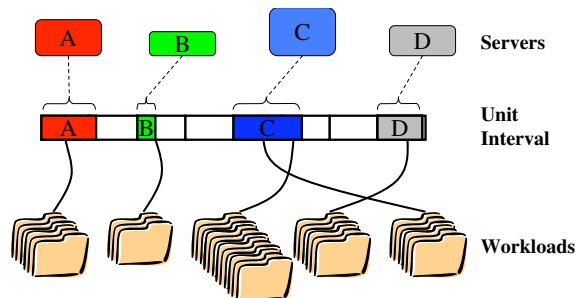


Figure 3. ANU randomization concept.

to read the data.

## 4 Load Distribution Protocol

We use a technique called adaptive resource selection (ARS) heuristic to distribute and balance I/O workload in a clustered storage system. The technique runs on client-side and obviates reconfigurations to existing server hardware/software infrastructures, which might be complicated and expensive, or sometimes even impracticable. Derived from ANU randomization [30], the ARS heuristic exploits data redundancy among storage server nodes to adaptively select appropriate nodes for I/O requests from a client. It does so on a per-request basis and provides short-term load balancing. It differentiates read and write operations and supports both replication and erasure coding schemes.

### 4.1 Basic Idea

The ARS heuristic builds on tunable hashing techniques derived from adaptive, nonuniform (ANU) randomization [30]. ANU randomization employs an extra level of abstraction – a unit interval as shown in Figure 3 – in the workload-to-server mapping. The unit-interval is partitioned into multiple equal-size non-overlapping subregions. ANU randomization maps each server to one or more subregions in the unit interval and hashes each workload unit to an offset in the unit interval. A server could occupy multiple non-contiguous subregions in the unit interval and completely occupies all but one subregion. We call the space in the unit interval occupied by a server its mapped region. A workload unit is then assigned to the server whose mapped region covers the workload unit's hashed offset. Based on performance updates reported by all servers, ANU randomization dynamically scales the server mapped regions to adjust workload distribution. It shrinks the mapped regions of overloaded servers to decrease their load and implicitly increases the mapped region sizes of other servers to gain load. In ANU randomization, server mapped regions do not occupy all the unit interval but instead occupy half of the unit interval.

The half occupancy is maintained as an invariant by ANU randomization to ensure that there is always an assignment of subregions satisfying the needs of all servers as well as an unassigned subregion available to a recovered server. Workload hashed to unoccupied regions in the unit interval is re-hashed until assigned to a mapped region. ANU randomization makes hash mapping tunable and has been shown to be an effective technique for load management of metadata operations in shared-disk file system clusters [29, 30]. The original ANU randomization scheme is not suitable for balancing I/O workload, as it is designed for metadata operations, which consist of little data and short-lived transactions.

The ARS heuristic adopts the same unit interval abstraction to distribute and balance I/O workload. It hashes each unit of workload – an I/O request to a logical volume in this case – to a unit interval that is fetched from a local pool of unit intervals (see Section 4.2). The unit interval has been partitioned among only the nodes that store a replica or an erasure-coded fragment of the requested data. The I/O request is then routed to the node for which the hash of that I/O request lies within its mapped region. Because data already exist at all the nodes, load balancing consists of request routing only. Ideally, nodes with relatively low load should be assigned larger mapped regions and thus larger share of requests are routed to them. To appropriately route I/O requests, the ARS heuristic periodically tunes node mapped regions using some client-observed performance metrics. We notice that tuning the unit interval data structure incurs no reconfiguration overhead such as data movement in this case. Given this fact and that I/O operations vary widely in size and completion time, some more complex performance metrics than the task latency used in ANU randomization are adopted to frequently tune the mapped regions. Those client-computed metrics measure a "stretched" I/O throughput and are described in detail in Section 4.3 and 4.4.

## 4.2 Algorithm Details

The ARS heuristic uses separate unit intervals for request routing among different set of nodes to avoid routing to a node that does not carry the requested data. It keeps all the unit intervals in a local pool for reuse to achieve incremental tuning and reduce reconstruction overhead, rather than creating a fresh copy of the unit interval data structure for each request routing and then deleting it. Each unit interval in the pool is partitioned among a unique set of nodes and used to route requests whose target data are exclusively stored on all nodes in the set. Each unit interval also contains a timestamp indicating the last time it was tuned. When it is to be used again for another request routing among the unique set of nodes, clients check its timestamp and re-tune it if necessary. If a client cannot find

the unit interval partitioned among a specific set of nodes from the pool, it simply creates one and then adds it into the pool. While ANU randomization requires a consistent unit interval data structure replicated on all clients, our algorithm does not require such synchronization and a client could keep quite a few different unit intervals in its pool.

The pool is implemented as a balanced binary search tree for efficiency. The key of each tree node is the sorted id's of the set of servers partitioning the unit interval stored at that tree node. For example, a client tries to access a logical data block. It queries the virtualization scheme and finds out that only servers C, B, and E store a copy or erasure-coded fragment of the data. Our algorithm then uses the lexically sorted list of the server id's, "BCE" in this case, as the key to locate the corresponding unit interval in the tree. We then compare its timestamp with the timestamps of the client-observed performance metrics of server C, B, and E to decide if a tuning to the unit interval is needed to make it up-to-date. If the tree search locates no matched unit interval, our algorithm creates a new one, with mapped region sizes of the servers proportional to their current client-observed performance metric values. If the tree size reaches an upperbound, a garbage collection process runs to delete oldest tree nodes based on the timestamp.

One of the most important features of the ARS heuristic is to efficiently support selection of multiple nodes from a single unit interval data structure. Because erasure-coding techniques require accessing multiple erasure-coded fragments (nodes) to retrieve the original data, an I/O request to a logical volume using erasure-coding redundancy actually consists of multiple I/O subrequests to be routed to multiple nodes. To achieve efficient multiple-node selection, we create an identical copy of the located unit interval data structure from the pool, and, realize multiple-node selection on the copy with the following steps:

- 1) We hash one of the unrouted I/O subrequests to the unit interval. If the subrequest is hashed to an unoccupied region, it is re-hashed using the next hash function from an agreed-upon family of hash functions until it is hashed to the mapped region of some server node.
- 2) The I/O subrequest is then routed to and thus served by the node whose mapped region the subrequest's hash offset falls into. And, that node is marked and its mapped region size shrinks to zero.
- 3) All unmarked nodes scale up their mapped regions to keep the half-occupancy invariant.

The above steps are repeated until all subrequests have been routed, i.e., the same number of nodes have been selected from the unit interval. Then, the copy is discarded. We define a single round execution of the above steps as

a selection round.

The above multiple-node selection procedure guarantees that selection of multiple nodes takes time linear to the number of nodes selected. Because the mapped region size of each selected node is set to zero, it effectively would not participate in the following rounds of selection. Therefore, only a previously unselected node could be selected in each round. Because the half-occupancy invariant is always maintained in the above procedure, on average two hash probes will locate a previously unselected node at each round of selection. Therefore, on average it totally takes  $2m$  hash probes to the unit interval data structure to select  $m$  nodes.

In either replication or erasure-coding based storage systems, writes must be serviced on all the nodes that store a replica or coded fragment of the data, while reads can be serviced by a subset of the nodes. Thus, we have different policies for read and write operations in our load distribution protocol. The different policies also adopt different performance metrics to evaluate node performance, because the same node could have quite different read and write performance. For example, most RAID-based systems deliver very good read performance. But write performance of these systems is generally not comparable with their read performance. Therefore, we also use two different unit interval data structures for the same set of nodes, one for read, one for write.

### 4.3 Read Policy

To read a replicated block, our protocol uses the ARS heuristic to adaptively select one node out of all the replica-bearing nodes to serve the request. The ARS heuristic simply hashes the LBN of the requested block to the unit interval, locates a node based on the hash offset, and then directs the read request to that node. With this approach, read requests to the same logical block are always hashed to the same offset and with high probability locate the same node. In other words, repeated reads to the same block could be directly served from server cache unless the serving node shrinks its mapped region and no longer covers that specific offset. Cache locality is a desirable property for storage systems, because it dramatically reduces I/O latency and makes efficient use of the limited cache space.

When reading multiple contiguous blocks of data, we use the LBN of the first logical block as the hash input to select the nodes. If the requested data lies across multiple logical segments, the original read request is divided into multiple read requests partitioned along logical segment boundary.

For each erasure-coded logical block, our protocol reads just  $m$  out of the  $n$  erasure-coded fragments, which is sufficient to reconstruct the original data. We use the multiple-

node selection procedure described above to select  $m$  nodes out of the  $n$  nodes. At each round of selection, we hash the concatenation of the requested block's LBN and a round counter  $r$  to the unit interval to select a node, where  $r$  is in  $[0, m-1]$  to represent the  $m$  I/O requests for reading  $m$  data fragments to reconstruct the original data. A large portion of the repeated reads could be directly served from server cache until significant changes of node's mapped regions. Multiple-block read of erasure-coded data is handled in the same way as that of replicated data, except the addition of the round counter to the hash input.

When applying the ARS heuristic to select nodes for read requests, we use the following equation

$$\frac{\sum \text{ReadSize}}{\sum (\text{ReadCompletionTime} - \text{ReadInitTime})}$$

to estimate each server node's read performance. The equation is based on client knowledge in which  $\text{ReadInitTime}$  is the time when a read request for replicated data or subrequest for erasure-coded data is sent from this client to the server,  $\text{ReadCompletionTime}$  is the time when server response to the (sub)request is received by the client minus the network latency between the client and the server, and  $\text{ReadSize}$  is the size of the requested data. The equation is computed over read request responses received by this client from the specific node since last computation. Our algorithm periodically re-computes this equation for each server. The computed value is then used as the performance metric to adjust node mapped region size.

The equation above measures both the throughput (disk speed, network bandwidth, etc.) and latency (I/O queue length) of a node. It is also simple to measure and calculate. The metric can be thought as "stretched" throughput over latency. When the I/O queue of a node is empty or short, the metric actually measures throughput of that node. As the queue gets longer, the metric value decreases and reflects the "stretched" throughput over I/O latency.

### 4.4 Write Policy

While write requests need to be served on all the nodes storing the data, we observe that write and commit may be separated. Write means sending data to a node to write asynchronously and the node responds as soon as the request has been enqueued, and, commit means writing data to persistent storage on a node. Commit is more expensive than write and guarantees the longevity of the data across node crashes. We assume that each node in the system supports both synchronous and asynchronous write operations.

When a client requests to write a replicated block, the write policy selects one node to commit the block. Because a single committed replica with the latest timestamp is

sufficient to recover the most recent data<sup>2</sup>, all other nodes will perform the less expensive write operation and return to the client as soon as the block is written to cache. In other words, the client sends a synchronous write request to the selected node and sends asynchronous write requests to all the other nodes that store a replica of the block. Similarly, when writing an erasure-coded block, a client dynamically selects  $m$  nodes to write synchronously and sends asynchronous write requests to the remaining  $n-m$  nodes.

Therefore, the write policy allows us to commit to less loaded nodes and write to the remaining (more heavily loaded) nodes. The selection of the more versus less expensive operation can be used to load-balance requests on write, even when writing to all the nodes. By writing synchronously to nodes with good write performance and asynchronously to other nodes, clients both ensure data reliability across node failures other than disk failure and experience improved write performance, because the write operation will not complete until the acknowledgement from the last node is received. If clients write synchronously to all the nodes carrying the data, an extremely busy or slow node will lead to significant performance loss.

Node selection in the write policy is also based on the ARS heuristic, as in the read policy. A similar equation is used to evaluate the write performance of a node:

$$\frac{\sum WriteSize}{\sum (WriteCompletionTime - WriteInitTime)} .$$

Note that for each node, the above equation is computed over *synchronous* write request responses from that node to accurately estimate its current disk write performance. Responses to asynchronous write requests are not used in the computation. The computed result is the write performance metric value for the node and a higher value indicates better disk write performance of a node. The ARS heuristic tunes the unit interval in a way that a higher value leads to a larger mapped region and an increased probability of being selected to perform a commit. The metric value could be sparse if there are no synchronous writes to that node from this client since the last computation. To deal with such cases, we update the write performance metric value for a node only if new synchronous write responses from that node are received by the client. Otherwise, we keep the value from last computation.

Our write protocol does not endanger data reliability when writing asynchronously to some nodes. The newly written data are lost only if all the nodes crash before flushing the cached data and nodes with committed data run into disk failures. Considering that the storage server nodes in our target system are located at different physical

<sup>2</sup>Requires a timestamp based data consistency protocol [9, 11]. It is out of the scope of this paper.

locations, the probability of such catastrophic failure is low.

#### 4.5 Discussion

By providing short-term load balancing, our protocol alleviates performance coupling of clustered storage servers. It adaptively distributes I/O workload among storage servers in a way that each storage server receives appropriate amount of workload. Therefore, slow servers in the system will not become a performance bottleneck and one can construct virtual volumes on top of servers that have highly different performances. Furthermore, because our algorithms make no assumption about hardware and network characteristics, it is free of vendor lock-in and facilitates assembling hardware from different vendors and over arbitrary networks. Lastly, because our protocol is based on client logic, there are no changes necessary to the existing storage server hardware, software, and network infrastructure.

Our protocol also includes an optimization to deal with transient network and server failures. Instead of reading from just the  $m$  selected nodes and after a timeout re-sending the request to the remaining  $n-m$  nodes because of not receiving enough responses, the optimization selects  $m+e$  nodes at the initial place under the presence of node or network failures.  $e$  is a dynamically adjusted variable based on current node responsiveness. It is set to 0 when all nodes responds to the client and set to small values such as 1 or 2 under transient node failures.

One challenge in distributed systems is to handle the "herds of requests" problem – many clients simultaneously try to access the few servers that are perceived as least loaded because the clients do not exchange information between each other. The unit interval abstraction used in our protocol addresses this problem. On each client, requests are hashed to the unit interval and then directed to the servers in proportion of their performance. Furthermore, there is no synchronization of global states or client decisions in our protocol to trigger this problem.

### 5 Performance Evaluation

We evaluate the performance of our load distribution protocol using simulations and compare it against three other protocols. Experimental results show that our protocol achieves comparable performance to a centralized protocol on system throughput and request latency. The simulation results also demonstrate that our protocol meshes well with both replication and erasure-coding data redundancy techniques and performs consistently under different inter-server latencies.

We compare the performance of our ARS heuristic based protocol with three other load distribution protocols in a simulated clustered storage system:

	# of disks	disk model	per-disk capacity	Disk RPM	Seek Time	Buffer Size	Interface	logical organization
<i>Node</i> <sub>0</sub>	2	Quantum Atlas 10K	9.1GB	10025	5ms	2MB	SCSI	N/A
<i>Node</i> <sub>1</sub>	9	IBM Ultrastar 18ES	9.1GB	7200	7ms	2MB	SCSI	RAID5
<i>Node</i> <sub>2</sub>	1	IBM Deskstar 7K400	400GB	7200	8.2ms	8MB	ATA-100	N/A
<i>Node</i> <sub>3</sub>	18	Seagate Cheetah 4LP	4.5GB	10033	7.7ms	512KB	Ultra SCSI	N/A

**TABLE I. The four simulated storage nodes in the clustered storage system.**

- a randomized protocol, which distributes an I/O request to randomly selected node(s) that store the data,
- a centralized protocol that has perfect knowledge of the system, and,
- an implementation of the D-SPTF protocol.

To improve performance, the randomized protocol and the centralized protocol read from or write synchronously to the minimum necessary number of nodes for each I/O request – 1 for replicated data and  $m$  for m-of-n erasure-coded data. They also write asynchronously to the remaining  $n - m$  nodes for write requests. The randomized protocol runs on client-side and randomly picks node(s) from the set of nodes carrying the data. The centralized protocol uses a central load distributor to distribute requests and has complete knowledge of the servers and workload. It uses a bin-packing approach to maximize throughput and presents an upperbound of load balance. While the original D-SPTF protocol balances reads only and writes all replicas to NVRAM to be flushed to disk later, our D-SPTF implementation treats writes the same as reads, because there is no NVRAM in our simulated storage servers. In our D-SPTF implementation, clients send each I/O request to all nodes that store the data. Prior to finishing serving the current request in disk, each node pre-selects the next request to serve and immediately sends a claim containing its service time bid for the pre-selected request to all other nodes that also receive the request. A node serves the pre-selected request only if until when the current request completes, no other nodes claim to serve the pre-selected request with a better bid. To have D-SPTF work with requests to m-of-n erasure coded data, we enforce a node to serve the pre-selected request if it does not receive at least  $m$  claims with better bids than its own bid when the current request completes. We note that while a pre-selected request to replicated data can be canceled by a single claim from another node, cancellation of a request to erasure-coded data requires at least  $m$  claims from other nodes.

In our experiments, we simulate a clustered storage system consisting of four heterogeneous storage server nodes, two heterogeneous logical volumes, and multiple clients. Each simulated node listens on a network port to receive client I/O requests and put them in a local queue. The DiskSim simulator [8] is used to simulate the storage subsystem of each node. The node's DiskSim

ID	Capacity	Segment Size	Block Size	Redundancy Type
0	100GB	128MB	64KB	3-way replication
1	200GB	512MB	64KB	2-of-4 erasure coding

**TABLE II. The two logical volumes constructed in the clustered storage system.**

thread fetches requests from the queue and serves them. Upon completion of a request, the DiskSim thread notifies the main thread, which then sends the response back to the client via TCP sockets. Table I shows the characteristics of the storage subsystems of the four nodes simulated. We disable the write-back cache of all simulated disks to enforce synchronous write for all commits. We maintain for each node a delayed write buffer to hold asynchronous writes, which is flushed upon disk idle or buffer full. Among the two simulated logical volumes, one (logical volume 0) employs replication to increase data reliability while the other (logical volume 1) uses erasure coding, as shown in Table II. Multiple clients are also simulated in our experiments. Each client includes a software module implementing the ARS heuristic. We also implement the storage virtualization scheme as a module in the client code to perform logical-to-physical address translations.

The experiments are driven by both synthetic and real-world I/O traces. The two synthetic I/O traces we use each consist of 10,604 requests from 5 clients in a 20-minute time interval, while one synthetic trace contains read requests only and the other one consists of both read and write requests with ratio 1:1. We call them read-only and mixed synthetic traces respectively. Request sizes of the two synthetic traces are both exponentially distributed with mean size of 8KB. Requested data of the synthetic traces are uniformly and randomly distributed in the logical volume spaces to avoid data locality among requests. We use the synthetic I/O traces to minimize cache effects to precisely evaluate the load-balancing performance of the four protocols. We also use a real-world financial I/O workload trace [26] to evaluate the performance of the protocols and verify the sanity of our synthetic traces. The one-hour financial workload we use from the trace consists of 249,496 I/O requests with average request size 15.5KB and 33% reads. In our simulations, unless otherwise noted, we evenly divide workload from the driven trace among the two simulated logical volumes.

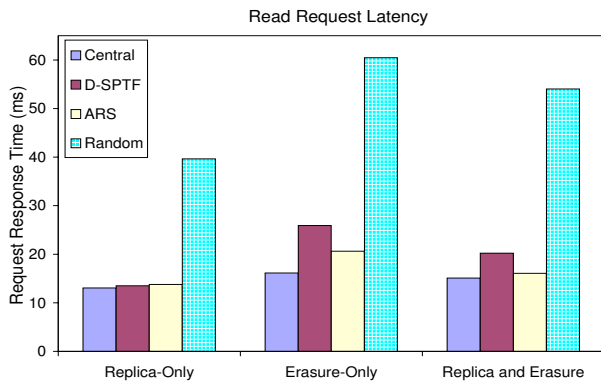


Figure 4. Response time for read requests.

To examine the efficiency of the protocols in distributing load, we measure the load balance metrics of the simulated system with each request distribution protocol. The metrics include average response time of client requests, which indicates if requests are distributed to the more capable servers when system is underloaded, and system throughput, which indicates if each node is assigned load proportional to its capacity under periods of high load or short-term load fluctuations. Because the original D-SPTF protocol balances read requests only, we first collect values of those metrics from experiments driven by read-only workload to find out the relative performance of the protocols to balance read requests. We then run experiments with mixed workload that contains both reads and writes to understand each protocol's capability to handle both read and write requests.

Figure 4 shows the average request response times of the four protocols. The experiments are driven by the synthetic read-only trace. To investigate how efficiently each protocol supports both the replication and erasure-coding redundancy schemes, we have each protocol distribute the reads under three different setups: all reads are for replicated data (i.e., logical volume 0); all reads are for erasure-coded data (i.e., logical volume 1); and, the reads are evenly split among the two logical volumes. The results in Figure 4 shows that our protocol performs closely to the centralized protocol in most cases. This indicates that our protocol efficiently distribute the requests to appropriate servers. The randomized protocol as expected performs the worst, because it distributes an equal number of requests to the heterogeneous servers and has no ability to tune the load distribution once the load is sent out. D-SPTF instead is able to tune load distribution in real-time by serving a request on the first-available server and canceling the request on other servers. Therefore, it achieves nearly ideal performance and outperforms our protocol when reading replicated data. However, Figure 4 also shows that it does

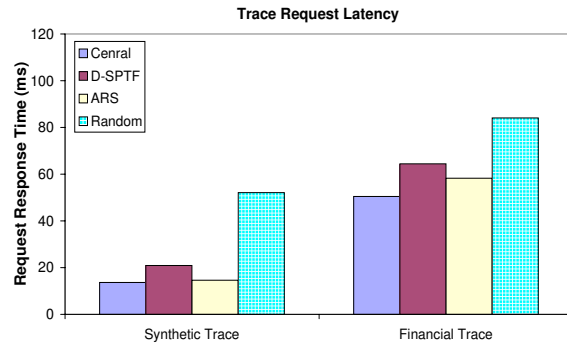


Figure 5. Response time for trace workload.

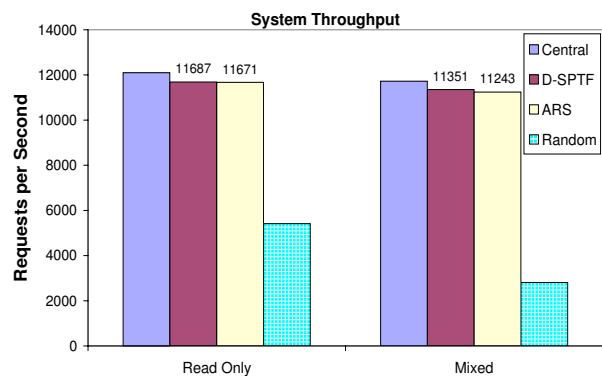
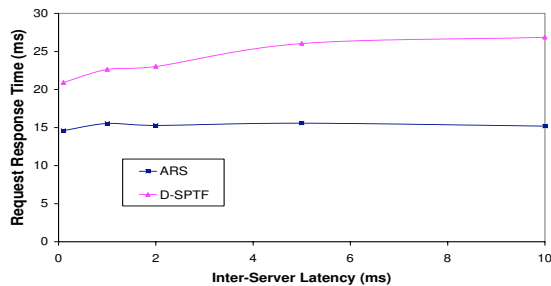


Figure 6. Throughput of the system.

not perform as well when reading erasure-coded data. This is because D-SPTF realizes its real-time load distribution tuning via fast inter-server messaging within a short time window. It is difficult for D-SPTF to cope with the high messaging overhead required to cancel a read to erasure-coded data.

Figure 5 shows the average request response times of both the mixed synthetic trace and the financial trace. Both traces consist of read and write requests that are evenly distributed to the two logical volumes. The financial trace result shows that our protocol efficiently balances both read and write requests and achieves performance comparable to the centralized protocol. The synthetic trace results in both Figure 4 and Figure 5 match the patterns of the real-world financial trace result and thus verify the sanity of the synthetic traces.

Figure 6 presents the throughput test results on the simulated system. We use a different workload for the throughput tests. In the throughput tests, we first send 50 requests to the system in a batch and then use a while loop to keep sending a new request upon receiving a completed one. In this way, we keep a constant number of



**Figure 7. Impact of inter-server latency to load balance.**

outstanding requests in the system to keep it busy to meet the maximum throughput capacity of each protocol, while avoiding overloading the system. The requests used in the throughput tests are randomly generated to avoid locality and minimize caching effects, except that the request sizes are exponentially distributed with mean size of 8KB. In one set of the tests, we generate random read requests only. In the other set, we generate both reads and writes with a ratio of 1:1. The experimental results are shown in Figure 6 and illustrate that both our protocol and D-SPTF achieve similar performance to the centralized protocol. Figure 6 also shows that D-SPTF achieves slightly better throughput than our protocol. This is because D-SPTF sends requests to all the nodes and thus the nodes are never idle. In contrast, our protocol distributes each request to a subset of the nodes and there could be some short periods when some node(s) sit idle.

We also run experiments with various network latencies to test the robustness of our protocol under different network conditions. We modified the IProute2 toolkit [15] to interact with the Linux `sch_delay` kernel module to inject artificial latency into inter-server communications. While all the previous experiments are conducted with the default one-way inter-server latency in our test environment, which is 0.1ms, we additionally test the average request response times of D-SPTF and our protocol with one-way inter-server latency set to be 1ms, 2ms, 5ms, and, 10ms respectively. We use the mixed synthetic trace workload to drive the tests and the measured average response times (minus network latency) are shown in Figure 7. The results show that our load distribution protocol works well with a wide range of inter-server latencies. This is because our protocol does not depend on inter-server communications to balance load. On the other hand, D-SPTF heavily relies on high-speed interconnects to aggressively exchange messages within a short time window among servers to balance each single request. Thus, the performance of D-SPTF degrades as the message latency increases.

## 6 Conclusion

We have described an adaptive load distribution protocol to provide short-term load balancing for I/O workload in clustered storage systems. It uses a decentralized tunable hashing scheme to dynamically distribute read and write requests to appropriate storage server nodes. It builds on client logic and requires no change to storage server software or hardware, or network. Thus, it facilitates the coupling of storage resources that are heterogeneous in many aspects. Our protocol makes no assumption about inter-server communication latency and works well under a wide range of network configurations. Experimental results verify the effectiveness of our protocol to balance I/O workload in clustered storage systems by showing that it performs closely to a centralized algorithm.

## References

- [1] Acopia Networks. <http://www.acopia.com/>.
- [2] A. Adya, W. Bolosky, M. Castro, R. Chaiken, G. Cermak, J. Douceur, J. Howell, J. Lorch, M. Theimer, and R. Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. In *OSDI: Symposium on Operating Systems Design and Implementation*. USENIX Association, 2002.
- [3] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch. Hippodrome: running circles around storage administration. In *Proceedings of the Conference on File and Storage Technologies*, 2002.
- [4] A. Brinkmann, M. Heidebuer, F. auf der Heide, U. Ruckert, K. Salzwedel, and M. Vodisek. V:drive - costs and benefits of an out-of-band storage virtualization system. In *Proceedings of the Twenty-First IEEE Conference on Mass Storage Systems and Technologies*, 2004.
- [5] P. Cao, S. Lin, S. Venkataraman, and J. Wilkes. The TickerTAIP parallel RAID architecture. *ACM Transactions on Computer Systems*, 12(3):236–269, 1994.
- [6] IBM Almaden Research Center. Intelligent Bricks, 2003. [http://almaden.ibm.com/StorageSystems/Advanced\\_Storage\\_Systems/Intelligent\\_Bricks](http://almaden.ibm.com/StorageSystems/Advanced_Storage_Systems/Intelligent_Bricks).
- [7] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke. The Data Grid: Towards an Architecture for the Distributed Management and Analysis of Large Scientific Datasets. *Journal of Network and Computer Applications*, 23:187–200, 2001.
- [8] CMU. The DiskSim Simulation Environment (Version 2.0), 2004. <http://www.pdl.cmu.edu/DiskSim/>.
- [9] S. Frolund, A. Merchant, Y. Saito, S. Spence, and A. Veitch. A decentralized algorithm for erasure-coded virtual disks. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'04)*, page 125. IEEE Computer Society, 2004.
- [10] G. Ganger, B. Worthington, R. Hou, and Y. Patt. Disk subsystem load balancing: Disk striping vs. conventional data placement. In *Proceedings of the Twenty-Sixth Annual Hawaii International Conference on System Sciences*, volume I, pages 40–49, 1993.
- [11] G. Goodson, J. Wylie, G. Ganger, and M. Reiter. Efficient byzantine-tolerant erasure-coded storage. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN'04)*, page 125. IEEE Computer Society, 2004.
- [12] Grid Physics Network. The GriPhyN Project, 2004. <http://www.griphyn.org/>.
- [13] A. Haeberlen, A. Mislove, and P. Druschel. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI'05)*, May 2005.
- [14] J. L. Hafner. Weaver codes: Highly fault tolerant erasure codes for storage systems. In *Proceedings of the FAST Conference on File and Storage Technologies*, 2005.

- [15] Iproute2 Utility Suite. <http://linux-net.osdl.org/index.php/Iproute2>.
- [16] J. Kubiatawicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gum-madi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS*. ACM, November 2000.
- [17] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 84–92, Cambridge, MA, 1996.
- [18] IP-based storage area networks. Technical Report available at – [http://www.lefthandnetworks.com/downloads/ip-san\\_wp.pdf](http://www.lefthandnetworks.com/downloads/ip-san_wp.pdf), Left-Hand Networks, 2002.
- [19] M. Luby. LT codes. In *The 43rd Annual IEEE Symposium on Foundations of Computer Science*, pages 271–282, 2002.
- [20] C. Lumb, R. Golding, and G. Ganger. D-SPTF: Decentralized request distribution in brick-based storage systems. In *Proceedings of ACM ASPLOS*, 2004.
- [21] M. Mitzenmacher. Digital fountains: A survey and look forward. In *IEEE Information Theory Workshop*, 2004.
- [22] OptIPuter system software framework. Technical Report available at – [http://www.cs.ucsd.edu/Dienst/UI/2.0/Describe/ncstrl.ucsd\\_cse/CS2004-0786](http://www.cs.ucsd.edu/Dienst/UI/2.0/Describe/ncstrl.ucsd_cse/CS2004-0786), UCSD, 2004.
- [23] M. O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *J. ACM*, 36(2):335–348, 1989.
- [24] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, and E. Zeidner. Internet small computer systems interface (iSCSI), 2004. <http://www.ietf.org/rfc/rfc3720.txt>.
- [25] SDSS. Sloan digital sky survey, 2004. <http://www.sdss.org/>.
- [26] UMass Trace Repository. <http://prisms.cs.umass.edu/repository/>.
- [27] H. Weatherspoon and J. Kubiatawicz. Erasure coding vs. replication: A quantitative comparison. In *Proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS)*, 2002.
- [28] S. B. Wicker and V. K. Bhargava. *Reed Solomon Codes and their Applications*. IEEE Press, 1994.
- [29] C. Wu and R. Burns. Handling heterogeneity in shared-disk file systems. In *Proceedings of the International Conference for High Performance Computing and Communications*, 2003.
- [30] C. Wu and R. Burns. Tunable randomization for load management in shared-disk clusters. *ACM Transactions on Storage*, 1(1):108–131, 2005.
- [31] H. Xia and A. Chien. RobuStore: Robust performance for distributed storage systems. Technical report, UCSD Technical Report CS2005-0838, 2005.
- [32] Z. Zhang and Q. Lian. Reperasure: Replication protocol using erasure-code in peer-to-peer storage network. In *Proceedings of SRDS*, 2002.