Towards Data-Driven I/O Load Balancing in Extreme-Scale Storage Systems

Sangeetha Banavathi Srinivasa

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science and Applications

Ali R. Butt, Chair
Nicholas Fearing Polys
Sharath Raghavendra

May 1, 2017
Blacksburg, Virginia

Keywords: Lustre storage system, Markov chain model, Max-flow algorithm,
Publisher-Subscriber, I/O load balance

Towards Data-Driven I/O Load Balancing in Extreme-Scale Storage Systems

Sangeetha Banavathi Srinivasa

(ABSTRACT)

Storage systems used for supercomputers and high performance computing (HPC) centers exhibit load imbalance and resource contention. This is mainly due to two factors: the bursty nature of the I/O of scientific applications; and the complex and distributed I/O path without centralized arbitration and control. For example, the extant Lustre parallel storage system, which forms the backend storage for many HPC centers, comprises numerous components, all connected in custom network topologies, and serve varying demands of large number of users and applications. Consequently, some storage servers can be more loaded than others, creating bottlenecks, and reducing overall application I/O performance. Existing solutions focus on per application load balancing, and thus are not effective due to the lack of a global view of the system.

In this thesis, we adopt a data-driven quantitative approach to load balance the I/O servers at extreme scale. To this end, we design a global mapper on Lustre Metadata Server (MDS), which gathers runtime statistics collected from key storage components on the I/O path, and applies Markov chain modeling and a dynamic maximum flow algorithm to decide where data should be placed in a load-balanced fashion. Evaluation using a realistic system simulator shows that our approach yields better load balancing, which in turn can help yield higher end-to-end performance.

That this work is sponsored in part by the NSF under the grants: CNS-1405697, CNS- 1422788, and CNS-1615411. Any opinions, findings, and conclusions expressed in this thesis are those of the author and do not necessarily reflect the views of NSF.

Towards Data-Driven I/O Load Balancing in Extreme-Scale Storage Systems

Sangeetha Banavathi Srinivasa

(GENERAL AUDIENCE ABSTRACT)

Critical jobs such as meteorological prediction are run at exa-scale supercomputing facilities like Oak Ridge Leadership Computing Facility (OLCF). It is necessary for these centers to provide an optimally running infrastructure to support these critical workloads. The amount of data that is being produced and processed is increasing rapidly necessitating the need for these High Performance Computing (HPC) centers to design systems to support the increasing volume of data.

Lustre is a parallel filesystem that is deployed in HPC centers. Lustre being a hierarchical filesystem comprises of a distributed layer of Object Storage Servers (OSSs) that are responsible for I/O on the Object Storage Targets (OSTs). Lustre employs a traditional capacity-based Round-Robin approach for file placement on the OSTs. This results in the usage of only a small fraction of OSTs. Traditional Round-Robin approach also increases the load on the same set of OSSs which results in a decreased performance. Thus, it is imperative to have a better load balanced file placement algorithm that can evenly distribute the load across all OSSs and the OSTs in order to meet the future demands of data storage and processing.

We approach the problem of load imbalance by splicing the whole system into two views: filesystem and applications. We first collect the current usage statistics of the filesystem by means of a distributed monitoring tool. We then predict the applications' I/O request pattern by employing a Markov Chain Model. Finally, we make use of both these components to design a load balancing algorithm that eventually evens out the load on both the OSSs and OSTs.

We evaluate our algorithm on a custom-built simulator that simulates the behavior of the actual filesystem.

Dedication

If an angel lived in this world

Mom, it would be you

No other person can be

So pure and true

I have never seen God

But He must be like dad

Who always makes me smile

And can’ t bear to see me sad

Thanks to both of you

For being so wonderful

I love you both for making

My life so beautiful

Acknowledgement

I would like to sincerely thank Dr. Ali R. Butt for his guidance and support throughout the project.

I would like to express my gratitude towards Dr. Nicholas Polys and Dr. Sharath Raghavendra for taking time out of their busy schedule and being on my committee.

I am highly indebted to Brian Marshall and Dr. James McClure for having provided me the support without which my journey at Virginia Tech would not have been smooth.

I would further like to thank Dr. Feiyi Wang and Dr. Sarp Oral for giving their valuable inputs about Lustre. I would also like to thank Ross G. Miller for testing my code on the test-bed and for helping me in my implementation of the monitoring tool.

My sincere gratitude to everyone at Distributed Systems and Storage Lab (DSSL) and Advanced Research Computing (ARC).

I would like to extend my gratitude towards the staff for helping me all along the way.

My deepest gratitude to my parents and sister for the strength they instilled in me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

High performance computing (HPC) is routinely employed in various science domains such as Physics, Geology, and Computational Biology, to simulate and understand the behavior of complex physical phenomena. These scientific simulations are resource intensive: they require both computing and I/O capabilities at scale. With the dawn of age of big data, there is a crucial need for especially revisiting the I/O subsystem of the extreme-scale computing systems to better optimize for and manage the increased pressure on the underlying storage systems [37, 38, 12]. This is made more complex by the bursty and unpredictable I/O patterns of applications, as well as the need to incorporate fault tolerance in the system via regular checkpoints.

Several factors affect the I/O performance. First, the number and kinds of applications that an extreme scale storage system should support is increasing rapidly [64], which leads to increased resource contention and creation of hot spots where some data or resources are consumed significantly more than others [9, 10]. Second, the underlying storage systems are often distributed , e.g., Ceph [60], GlusterFS [16], and Lustre [17], and adopt a hierarchical design comprising thousands of distributed components connected via complex networks. Managing and extracting peak performance from such resources is non-trivial and require careful design. With changing application characteristics, static approaches (e.g., [25, 58]) to make these design decisions are not sufficient,

necessitating dynamic solutions [11]. Third, the storage components can develop load imbalance across the I/O servers, which in turn impacts application performance and time to solution. Consequently, achieving load balancing in the storage system is a key requirement for achieving a sustainable solution, especially for the forthcoming exascale era.

Load balancing for extreme-scale storage systems has been the focus of a number of recent works [26, 25]. Extant systems typically attempt to perform load balancing by either having limited support for read shedding to redirect read requests to replicas of the primary copy, e.g., in Ceph [44], or performing data migration. Alternatively, per application load balancing has also been considered to balance the load of an application across the various I/O servers [58]. Shortcomings of such approaches include a lack of a global view of the system, and a focus on a small subset of metrics (e.g., the storage capacity in case of migration). Thus, these approaches cannot guarantee that the aggregate I/O load of multiple applications concurrently executing and accessing a parallel file system (including applications with bursty behavior) is evenly distributed under typical real-world scenarios.

Consider the Lustre file system that forms the backend storage of many of the HPC centers that host the Top 500 supercomputers [27]. The default strategy of Lustre to allocate storage targets to I/O requests follows a round-robin approach. Experiments show that this approach is inclined to either under- or over-utilize the resources. Lustre is a hierarchical parallel file system technology involving components in different layers. State of the art solutions to load balance the storage system utilize the access frequency [58] of components in multiple layers for a particular job while performing I/O. While offering a significant improvement over standard Lustre, using frequency information alone is insufficient to provide a load-balanced system across varying applications [8, 14, 13]. The primary reason is that frequency with which a component in a particular layer is accessed is not truly representative of the load of that component. Another issue is that such solutions focus on per-application load rather than the global load balancing of the storage system.

In this thesis, we address the load imbalance problem by enabling a global view of the statistics of key components in Lustre. We go beyond just network load balancing, e.g., as in NRS [48], to ensure that the Lustre Object Storage Targets (OSTs) that actually store and serve the data along with other I/O system components are load balanced. As not all parameters contribute to

the load of a particular component, we design an efficient monitoring tool that can dynamically capture the system load at run time. We employ a Markov chain model to learn the behavior of the application from the past application requests, as well as to predict future requests. This information is then used by a dynamic maximum flow algorithm to enable a global mapper to evenly distribute application I/O load to the OSTs. To this end, the mapper creates and maintains a dynamic candidate list of OSTs, which is then used to assign OSTs to service I/O request in a way that is expected to keep the load balanced across the targets. The proposed global mapper co-locates with Lustre's Metadata Server (MDS) that has a global view of all other components. Thus, we do not introduce any additional performance or scalability bottlenecks in the system, rather exploit the existing hierarchy to our advantage.

1.2 Related Work

Load management has been incorporated into a number of modern distributed storage system designs. GlusterFS [16] uses elastic hashing algorithm that completely eliminates location meta-data to reduce the risk of data loss, data corruption, and data unavailability [32] and reduces the load on any one server, however, no load balancing is supported across the storage targets. Ceph [60, 44] uses dynamic load balancing based on CRUSH [61], a pseudo-random placement function. It also adds limited support for read shedding, where clients belonging to a read flash crowd are redirected to replicas of the primary copy of the data instead of the primary copy itself. However, the approach balances the read load and does not guarantee that the primary copies themselves are evenly distributed for optimum utilization of the storage resource. In contrast, our approach considers multiple factors and uses a global view of the system to make load balancing decisions for storage targets.

Several recent storage systems have explored optimization techniques for load balancing. In [25], dynamic data migration is proposed to balance the load under various constraints using constraint satisfaction search. Such approaches add the overhead of migration, while also servicing the need to maintain availability and consistency. The VectorDot [54] algorithm is able to incorporate all these different constraints, as it is a multidimensional knapsack problem. It is also suitable for hi-

erarchical storage systems, as it can model hierarchical constraints. However, unlike our approach, the algorithm cannot effectively use crucial history information for load balancing.

Machine learning and data mining techniques have also been used for load balancing (and for the more general problem of resource allocation). Martinez et. al. [42] introduce basic learning techniques for improving scheduling in hardware systems. These techniques are focused on individual hardware components and cannot be easily adapted to distributed file systems. A rule based approach to balance load in distributed file servers using graph mining methods is proposed in [31]. The strategy first identifies the access patterns of files in the system. This information is used to relocate the file sets among different file servers. Schaerf et. al. [51] explore the problem space of adaptive load balancing using reinforcement learning techniques. These works are complementary to our approach and provide a large design space, but require significant effort in feature selection and experimentation to enable such techniques in our target problem and workloads. We leverage such works in our approach to realize better I/O behavior capturing and improved predictions.

k-server problem, an online algorithm must control the movement of a set of k servers, represented as points in a metric space, and handle requests that are also in the form of points in the space [3]. As each request arrives, the algorithm must determine which server to move to the requested point. The goal of the algorithm is to keep the total distance all servers move small, relative to the total distance the servers could have moved by an optimal adversary who knows in advance the entire sequence of requests [24] [28]. A variation of this approach has been made use of in formulating the Dynamic Distributed Load Balancing (DDLB).

Finally, Google has recently started exploring using machine learning to optimize various system metrics. For example, deep learning is used in order to efficiently predict the Power Usage Effectiveness (PUE) of nodes in a cluster [30, 43], where 19 features are selected to train a neural network. The basic idea is to run the algorithm on training data by using random model parameters and use a forward propagation algorithm to find the cost function as the square error between predicted and actual values. Then back propagation is performed, wherein the error value learnt in the cost function is propagated back to each layer and the model is updated. While these techniques show the promise of machine learning for optimizing system parameters, they do not directly apply to load balancing in storage systems.

1.3 Contribution

Following are the main contributions of our work:

1. We provide a distributed data monitoring tool that can be integrated with any Lustre deployment. This tool is capable of collecting the usage statistics from each of the layers.
2. We provide a mechanism to predict the application I/O pattern which could prove useful in different scenarios such as in the design of extant file systems in HPC centers.
3. The load balancing model proposed will help in optimizing the load across the file system and will in turn benefit the many supercomputing facilities that deploy Lustre.
4. We have proposed an ingenious methodology wherein we have applied ideas learned in machine learning to solve a problem in parallel file systems.
5. We also evaluate the effectiveness and scalability of our approach. Experiments show that our approach helps in achieving a better load-balanced storage servers, which in turn can yield higher end-to-end system performance.

1.4 Document Overview

The document is organized as follows.

Chapter 2 lays down the background necessary to understand the implementation. This chapter starts off by giving an overview of the Lustre file system architecture. We then move on to studying the details of Titan architecture. From there, the chapter proceeds to explaining the basics of Markov Chain Model. This Model has been made use of at a later stage in our implementation. Finally, an overview of I/O Performance statistics is presented. This again will be referenced during the design of the Distributed Data Monitoring tool.

Chapter 3 discusses the Distributed Data Monitoring Tool in detail. The chapter starts off with a brief overview and progresses into explaining the design of the tool. The chapter then discusses

about the various features of the tool discussing simultaneously about the various design decisions that were made based on the constraints of the system.

Chapter 4 explains the approach taken to predict the applications' I/O behavior. As discussed earlier, we are presented with variability from the application front. This chapter explains the design and implementation approach of the library developed and integrated in MDS, in order to predict the applications' behavior.

Chapter 5 focuses on the load balancing model. As a final step, we design the load balancing algorithm. The chapter briefly explains about the different metrics and constraints that are accounted for in designing the algorithm. The chapter then explains about a single iteration in the algorithm. This groundwork is then used to construct the Max-flow graph.

Chapter 6 is the Evaluation of our algorithm. We evaluate our work on a custom-built simulator. We compare our results with a heuristic model and the Round-Robin model. Results of the scalability study is presented towards the end of the chapter.

Chapter 7 summarizes the work and presents some future work.

Chapter 2

Background

2.1 Lustre File System

2.1.1 Lustre Architecture

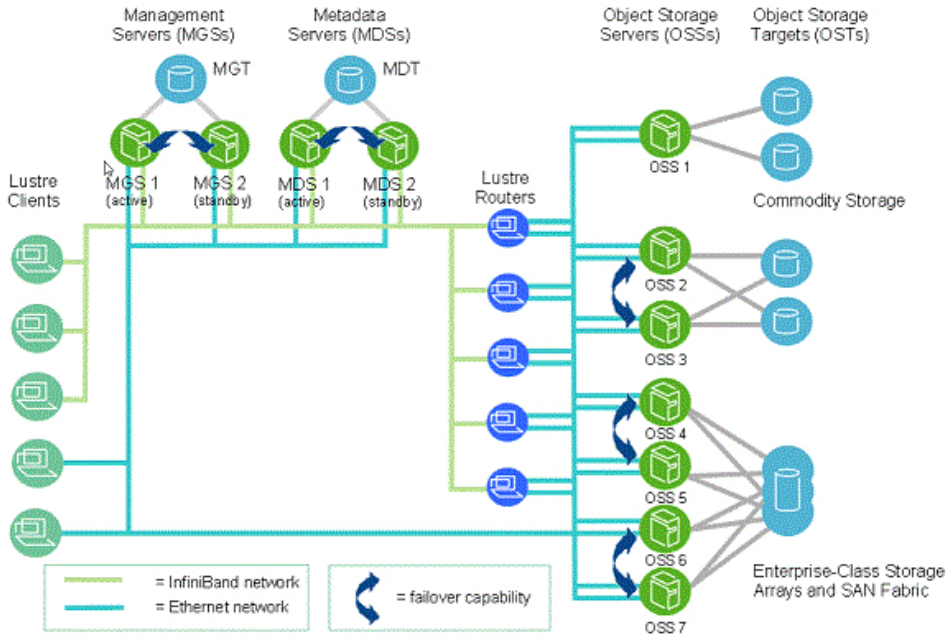

Figure 2.1: Lustre File system. [36]

Lustre architecture shown in Figure 2.1 is a storage architecture for clusters. The central component of Lustre architecture is the Lustre file system, which is supported on the Linux operating system and provides a POSIX standard-compliant UNIX file system interface [33].

Lustre storage architecture is used for many different kinds of clusters. It is best known for powering many of the largest high-performance computing (HPC) clusters worldwide, with tens of thousands of client systems, petabytes (PB) of storage and hundreds of gigabytes per second (GB/sec) of I/O throughput.

A Lustre file system consists of a number of machines connected together and configured to service file system requests. The primary purpose of a file system is to allow a user to read, write, lock persistent data. Lustre file system provides this functionality and is designed to scale performance and size as controlled, routine fashion.

In a production environment, a Lustre file system typically consists of a number of physical machines. Each machine performs a well defined role and together these roles go to make up the file system.
A typical Luster installation consists of:

- One or more clients.
- A Meta-data service.
- One or more Object Storage services.

1. **Client**

   A client in the Lustre file system is a machine that requires data. This could be a computation, visualization, or desktop node. Once mounted, a client experiences the Lustre file system as if the file system were a local or NFS mount.

2. **Meta-data services**

   In the Lustre file system meta-data requests are serviced by two components: the Meta-data Server (MDS, a server node) and Meta-data Target (MDT, the HDD/SSD that stores the meta-data).

   All a client needs to mount a Lustre file system is the location of the MDS. Currently, each Lustre file system has only one active MDS. The MDS persists the file system meta-data in the MDT.

   - ***Meta-data Servers (MDS)*** The MDS makes meta-data stored in one or more MDTs available to Lustre clients. Each MDS manages the names and directories in the Lustre file system(s) and provides network request handling for one or more local MDTs.
   - ***Meta-data Targets (MDT)*** For Lustre software release 2.3 and earlier, each file system has one MDT. The MDT stores meta-data (such as filenames, directories, permissions and file layout) on storage attached to an MDS. Each file system has one MDT. An MDT on a shared storage target can be available to multiple MDSs, although only one can access it at a time. If an active MDS fails, a standby MDS can serve the MDT and make it available to clients. This is referred to as MDS fail-over.

     Since Lustre software release 2.4, multiple MDTs are supported in the Distributed Name-space Environment (DNE). In addition to the primary MDT that holds the file system root, it is possible to add additional MDS nodes, each with their own MDTs, to hold sub-directory trees of the file system.

     DNE also allows the file system to distribute files of a single directory over multiple MDT nodes. A directory which is distributed across multiple MDTs is known as a striped directory.

3. **Object Storage services**

   Data in the Lustre file system is stored and retrieved by two components: the Object Storage Server (OSS, a server node) and the Object Storage Target (OST, the HDD/SSD that stores the data). Together, the OSS and OST provide the data to the Client.

   A Lustre file system can have one or more OSS nodes. An OSS typically has between two

and eight OSTs attached. To increase the storage capacity of the Lustre file system, additional OSTs can be attached. To increase the bandwidth of the Lustre file system, additional OSS can be attached.

- ***Object Storage Servers (OSS***) The OSS provides file I/O service and network request handling for one or more local OSTs.
- ***Object Storage Target (OST)*** User file data is stored in one or more objects, each object on a separate OST in a Lustre file system. The number of objects per file is configurable by the user and can be tuned to optimize performance for a given workload.

4. **Networking** Lustre uses LNET to configure and communicate over the network between clients and servers. LNET is designed to provide maximum performance over a variety of different network types, including InfiniBand, Ethernet, and networks inside HPC clusters such as Cray XP/XE systems [1].

2.1.2 I/O Flow in Lustre File System

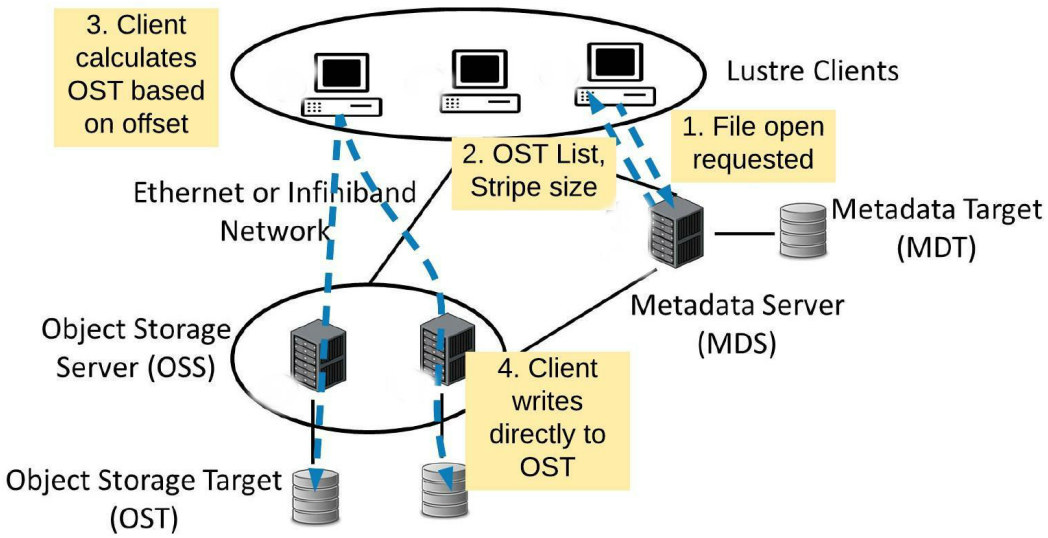

Figure 2.2: Lustre client requesting file data.

The below steps are illustrated in Figure 2.2

- When the client opens a file, it sends a request to the MDS server .
- The MDS server responds to the client with information about how the file is striped (which OSTs are used, stripe size of file, etc.).
- Based on the file offset, client can calculate which OST holds the data.
- Client directly contacts appropriate OST to read/ write data.

2.1.3 File Striping in Lustre File System

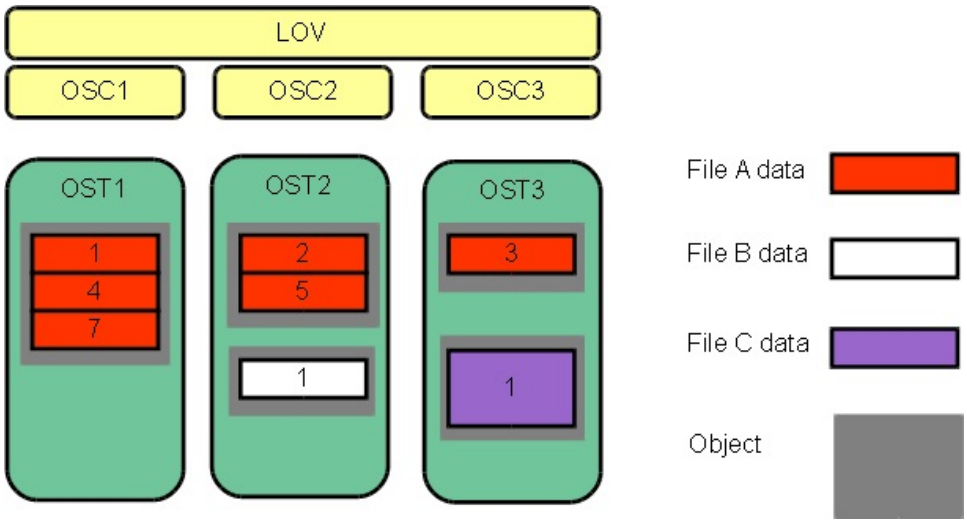

Figure 2.3: File striping on a Lustre file system. [36]

One of the main factors leading to the high performance of Lustre file systems is the ability to stripe data across multiple OSTs in a Round-Robin fashion. Users can optionally configure for each file the number of stripes, stripe size, and OSTs that are used.

Striping can be used to improve performance when the aggregate bandwidth to a single file exceeds the bandwidth of a single OST. The ability to stripe is also useful when a single OST does not have enough free space to hold an entire file.

In the Lustre file system, a RAID 0 [21] pattern is used in which data is ”striped” across a certain number of objects as shown in Figure 2.3. The number of objects in a single file is called the `stripe_count`.

Each object contains a chunk of data from the file. When the chunk of data being written to a particular object exceeds the `stripe_size`, the next chunk of data in the file is stored on the next object.

Default values for `stripe_count` and `stripe_size` are set for the file system. The default value for `stripe_count` is 1 stripe for file and the default value for `stripe_size` is 1MB. The user may change these values on a per directory or per file basis.

Although a single file can only be striped over 2000 objects, Lustre file systems can have thousands of OSTs. The I/O bandwidth to access a single file is the aggregated I/O bandwidth to the objects in a file, which can be as much as a bandwidth of up to 2000 servers. On systems with more than 2000 OSTs, clients can do I/O using multiple files to utilize the full file system bandwidth.

2.2 Titan Architecture

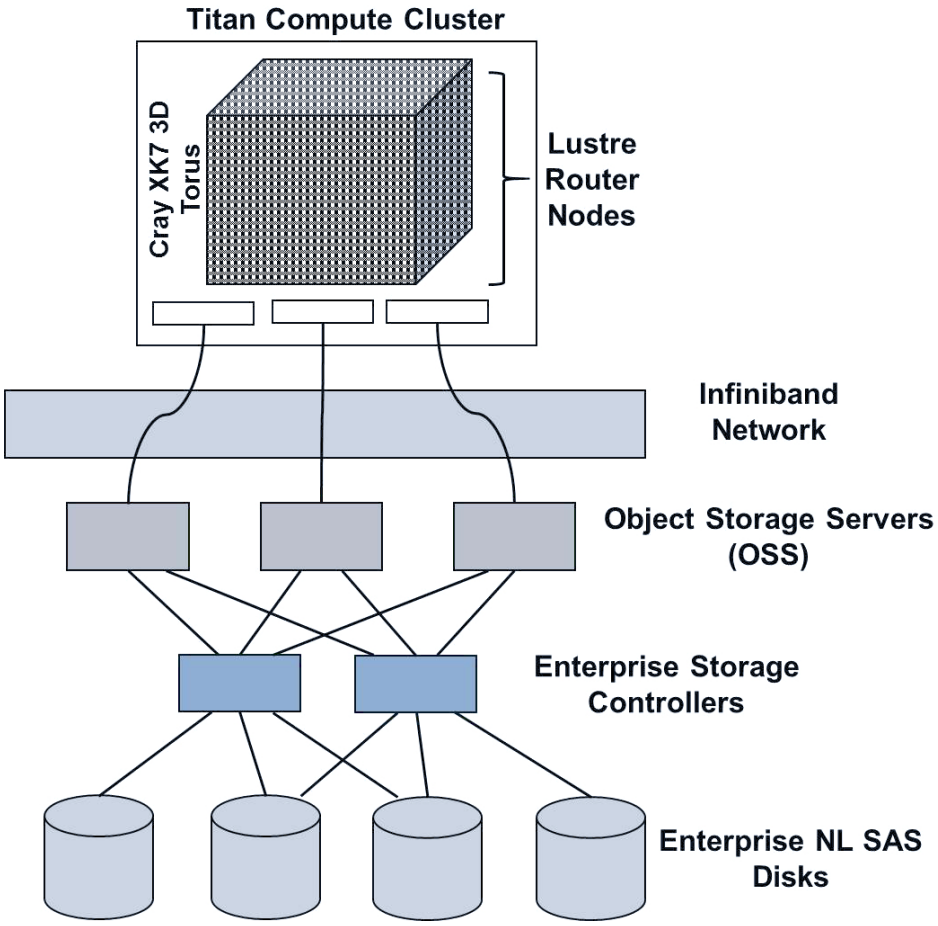

Figure 2.4: Overview of the considered HPC system.

Titan is a Cray XK7 system with 18,688 compute nodes, 710 TB of total system memory [58]. This high capability compute machine is backed by a center-wide parallel file system known as Spider II[23]. Spider II is one of the worlds fastest and largest POSIX complaint parallel file systems. It is

designed to serve write-heavy I/O workloads generated by Titan compute clients and other OLCF resources. The topology and architecture details of Spider II infrastructure are illustrated in Figure 2.4 and described as follows:

On the back-end storage side, Spider II has 20,160 disks organized in RAID 6 arrays. Each of these RAID arrays act as a Lustre Object Storage Target (OST). An OST is the target device where the Lustre parallel file system does file I/O (read or write) at the object layer. These OSTs are connected to Lustre Object Storage Servers (OSSes) over direct InfiniBand FDR links; these OSSes currently run Lustre parallel file system version 2.4.2. There are a total of 288 OSSes and each OSS has 7 OSTs (a total of 2,016 OSTS). Spider II is configured and deployed as two independent and non-overlapping file systems to increase reliability, availability, and overall meta data performance. Each file system, therefore, has 144 Lustre OSSes and 1,008 Lustre OSTs.

Each OSS is connected to a 36-port IB FDR Top Of the Rack (TOR) switch. Each TOR switch is connected with a total of 8 OSSes. Each switch also connects to two 108-port aggregation switches. The aggregation switches provide connectivity for the Lustre meta data and management servers.

On the front-end at the compute side, there are two different types of nodes in Titan: compute and Lustre I/O router nodes. Both types of nodes are part of the Gemini network[20] in 3D torus topology[7]. Each node has a unique network ID (NID) for addressing purposes. A total of 440 nodes on Titan are configured as Lustre I/O routers, 432 of which are used for routing Lustre file I/O traffic between Titan and Spider and 8 are used for routing Lustre meta data I/O. Titan I/O routers are connected to SION TOR switches via InfiniBand FDR links. Note that the SION TOR switches enable these I/O routers to reach to the back-end storage system (OSSes and OSTs).

2.3 Markov Chain Model

Markov chains, named after Andrey Markov, are mathematical systems that hop from one ”state” (a situation or set of values) to another [47]. For example, if we made a Markov chain model of a baby’s behavior, we might include ‘playing’, ‘eating’, ‘sleeping’, and ‘crying’as states, which together with other behaviors could form a ‘state space’: a list of all possible states. In addition,

on top of the state space, a Markov chain tells the probability of hopping, or 'transitioning', from one state to any other state—e.g., the chance that a baby currently playing will fall asleep in the next five minutes without crying first [5].

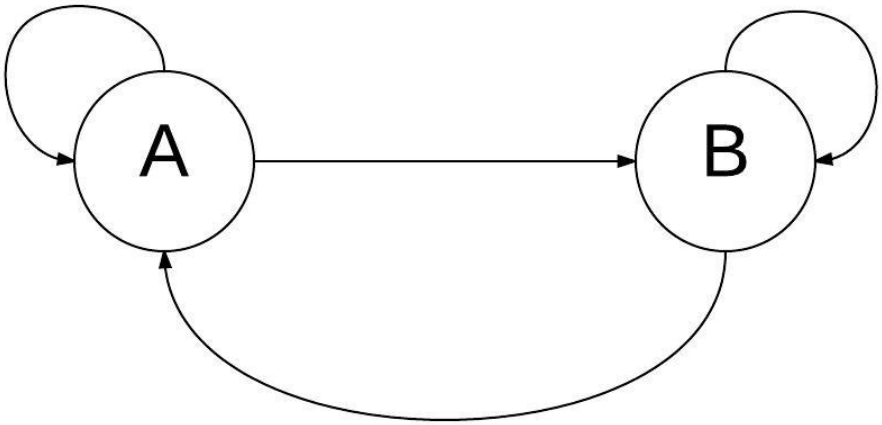

Figure 2.5: Illustration of Markov Chain Model.

With two states (A and B) in our state space, there are 4 possible transitions (not 2, because a state can transition back into itself) depicted in Figure 2.5. If we're at 'A' we could transition to 'B' or stay at 'A'. If we're at 'B' we could transition to 'A' or stay at 'B'. In this two state diagram, the probability of transitioning from any state to any other state is 0.5.

Markov chain is a simple concept which can explain most complicated real time processes. Speech recognition, Text identifiers, Path recognition and many other Artificial intelligence tools use this simple principle called Markov chain in some form [2].

Markov chain is based on a principle of memorylessness [2]. In other words, the next state of the process only depends on the previous state and not the sequence of states. This simple assumption makes the calculation of conditional probability easy and enables this algorithm to be applied in a number of scenarios.

Component	Factors	Discussion

Metadata Server (MDS)	`CPU%`
	`Memory%`
	
	`/proc/sys/lnet/stats`
		CPU and memory utilization reflect the system load.
		
		Load on the Lustre networking layer connected to MDS.

Metadata Target (MDT)	`mdt.*.md_stats` `mdt.*.job_stats`	Overall metadata stats per MDT. Metadata stats per MDT per job.

Object Storage Server (OSS)	`CPU%`
	`Memory%`
	
	`/proc/sys/lnet/stats`
		Reflects the system load of the management server.
		
		Load on the Lustre networking layer connected to OSS.

Object Storage Target (OST)	`obdfilter.*.stats`
	`obdfilter.*.job_stats`
	`obdfilter.*OST*.kbytesfree`
	`obdfilter.*OST*.brw_stats`
		Overall statistics per OST.
		Statistics per job per OST.
		Available disk space per OST.
		I/O read/write time and sizes per OST.

Table 2.1: I/O performance statistics for relevant system components.

2.4 I/O Performance Statistics

As Lustre is a hierarchical system, we identify the factors that affect the I/O performance in every layer of the system as shown in Table 2.1. Whenever a file creation request arrives at the MDS, the MDS provides to the Lustre client a list of OSTs on which the file is to be created. Data in Lustre is stored and retrieved by two components: OSS and OST. Therefore, the servers (MDS and OSS) play a role in all the I/O operations. I/O performance thus depends on the load on the servers. Factors that play a direct role in the system load on MDS and OSS are: CPU utilization, memory usage, and network usage (network usage is further discussed below). The values of these factors can be extracted using the Unix `sar` system monitoring tool [39].

I/O performance also depends on the rate of I/O operations that are performed on OSTs. This factor depends on the proportion of read and write load per OST. The higher the percentage of write operations, the lower is the I/O operations per second. The values of these factors can be extracted using the `obdfilter.*.stats` (provided by Lustre to report data per OST). Another important factor that affects the load on OSTs is the available disk space on a particular OST (`obdfilter.*OST*.kbytesfree`). In order to know the I/O operations performed by a particular job on a given OST, we need to examine the `obdfilter.*.job_stats`. I/O load on an OST is also affected by the time taken to perform I/O operations as well as the sizes of the I/O requests. These factors can be retrieved from `obdfilter.*OST*.brw_stats`.

The path of each met-adata I/O operation also includes accessing the MDT. Thus MDT load can also affect the meta-data I/O performance. The total I/O load on this component can be retrieved from `mdt.*.md_stats`, which gives the overall statistics of meta-data stored for all the jobs. To retrieve per-job meta-data statistics, we can utilize `mdt.*.job_stats`.

The performance for serving meta-data and the I/O rate of serving the actual data to the clients is dependent on the network performance. An overly congested network affects the I/O performance adversely. Thus, the read/write network bandwidth of the servers (MDS and OSS) plays an important role in load balancing the system. The values for read/write network bandwidth can be extracted using the `lst stat` interface (`/proc/sys/lnet/stats`), which runs over LNET and Lustre network drivers.

Lustre I/O performance is also affected by OST performance variability. Xie et. al. [62] show that slow OSTs (i.e., *stragglers*) can result in severely limited bandwidth for parallel coupled writes. Stragglers also limit striping bandwidth, thus reducing the benefits of parallelism. A comprehensive load balancing scheme for Lustre should, therefore, account for the effect of stragglers and assign load to the slow-performing OSTs accordingly.

Our approach extracts the above mentioned system statistics, and leverages this information to better load balance the storage system.

Chapter 3

Distributed Data Monitoring Tool

3.1 Overview

Distributed Data Monitoring Tool (DDMT) is designed to run across Lustre file system as shown in Figure 3.1. The tool is responsible for collecting usage statistics from each of the layers and these statistics are finally collated at MDS.

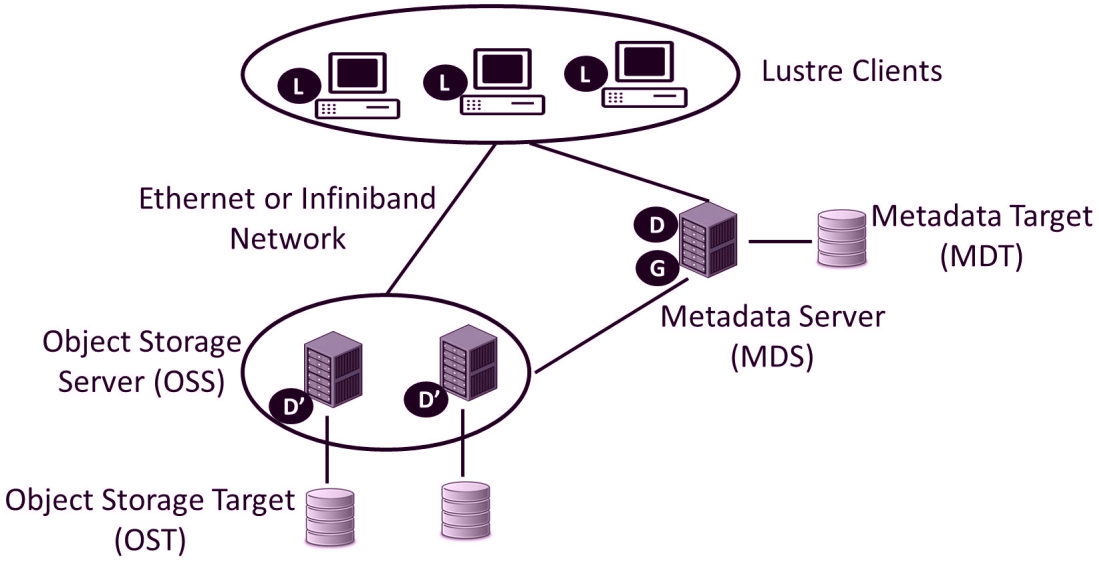

Figure 3.1: Design of DDMT. G is the subscriber. D and D' are publishers.

3.2 Design

The tool can be broadly viewed as a set of publishers and subscriber processes running on different Lustre components. Publishers probe the file system and make use of various Lustre tools like obdfilter to collect the current usage as well as performance statistics.

Publishers on each of the layer collect specific metrics, collate them and send them up to the next layer.

OST related statistics are collected first and are then sent up the hierarchy to OSS. At OSS, load of OSS is retrieved at each instant and this measure along with other OSS related statistics is collected. MDT and MDS publishers work in a similar way. Finally, OSS and MDS publishers send out the combined collected statistics to the subscriber that is running on the MDS instance.

This way all the important metrics are brought up till the MDS for further processing.

3.3 Approach

In order to implement the tool several design decisions had to be made. The final tool is the most optimal implementation among the various design strategies considered.

3.3.1 Design trade-off between Socket Programming based approach and Message Broker approach

The very first implementation that one can intuitively come up with based on the above design description is the socket programming based approach as depicted in Figure 3.2. This is exactly the approach that was taken in the initial implementation of the tool.

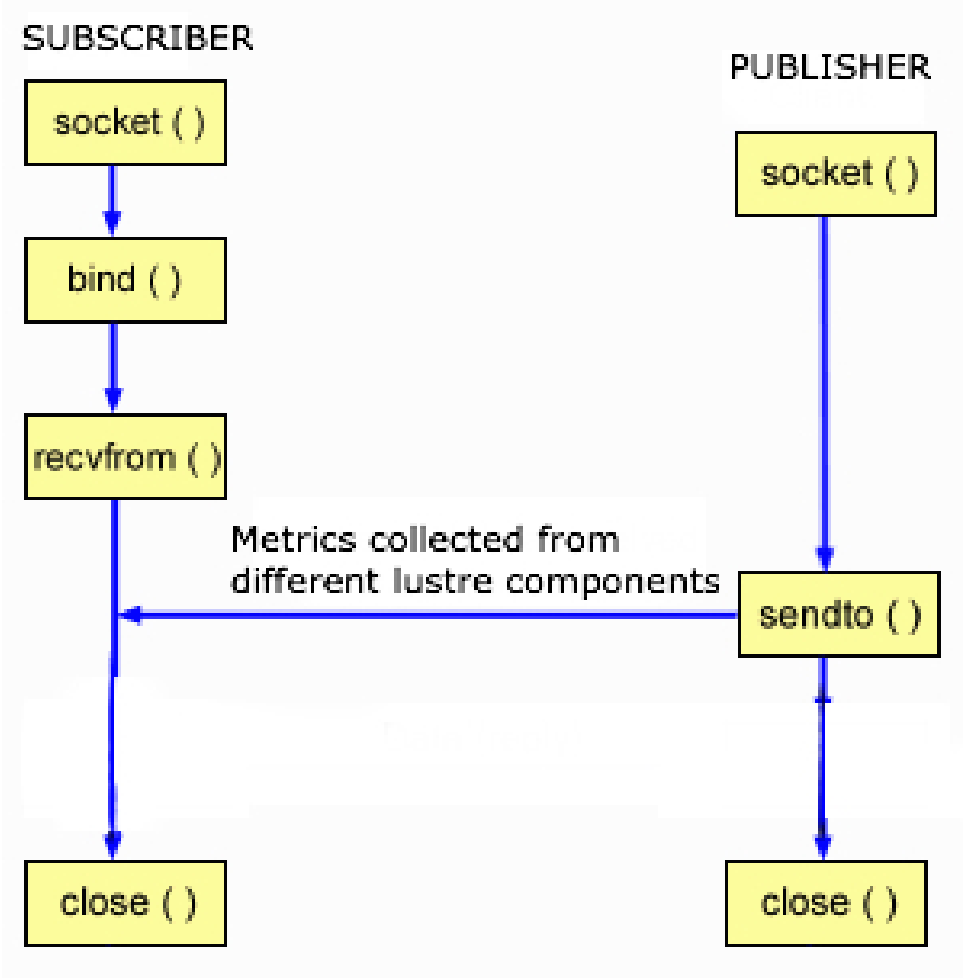

Figure 3.2: Publishers sending across collected metrics to the subscribers.

The subscriber on the MDS was configured in such a way that it would wait for the publisher to connect on a specific port and then it would listen on that port till the publisher wrote out the statistics collected at each instant of time onto a buffer.

The publisher that was running on each of the Lustre components collected the metrics and wrote them out to the subscriber.

This approach works well when we need not poll very frequently and when the Lustre file system is running on a small set of nodes. But when the file system is scaled out and it runs on a larger number of nodes like in most production systems, the packets that are being written out tend to be lost and it also adds extra load on the MDS as there will always be a process listening for packets

to arrive. This can become a bottleneck as MDS will become unresponsive to client requests and can thus slow down the whole system. This problem can further get aggravated when the packets that are being sent out get dropped due to buffer overflow.

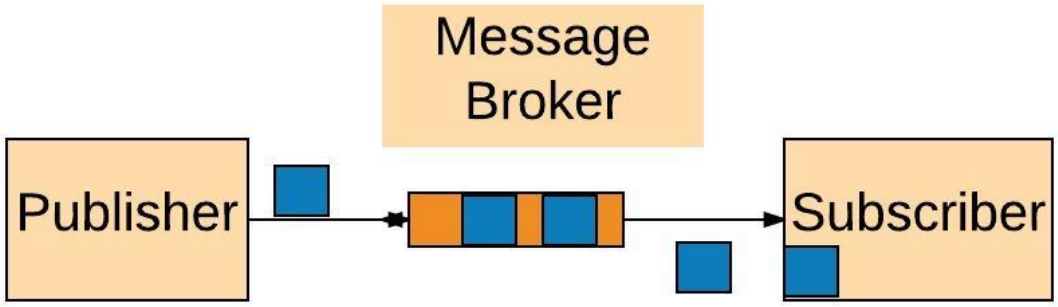

Figure 3.3: Publishers sending across collected metrics to the subscribers via message broker.

In order to overcome these issues, message broker implementation was considered as an alternative as depicted in Figure 3.3. In this approach we made use of RabbitMQ which is a leading message broker being used in many industries and HPC centers these days. The idea behind a message broker is simple. We can basically assume that there is a FIFO buffer queue onto which each of the publishers connect to and write out the message/packet they have at a point in time. These packets can then be read out by the subscriber as and when it is less lightly loaded. This way, the problem of losing packets is eliminated. Also, the MDS is no longer a bottleneck and it can retrieve the packets after it has serviced the client requests. This way our solution can easily scale and the publishers and subscribers can exchange messages comprising of usage and performance statistics in an elegant manner.

3.3.2 Back-end collected database to store the collected statistics

MySQL[45] is a legacy database and it is very easy to use as most systems provide support for this database.

Statistics are being collected from each of the layers and within each layer many different metrics

are being collected. We come up with a database in which there would be specific table dedicated to each of the metric and these tables are indexed by the index of the specific component from which it has been collected. Say, for example if the statistics comes from OST000001 then the index would be just that. The same would be done for statistics collected from other layers as well.

This implementation can be made use of by providing a configuration file to the tool at startup. This support has been retained in the final implementation in order to support traditional systems that make use of MySQL database.

MySQL database suffers from issues related to scalability and it is not tolerant towards changes made to the schema. Databases that are running on such a scale do undergo changes over time and having a rigid schema makes it less adaptive to changes. The access time over very large databases is also very long. Thus, many systems now make use of NoSQL databases like MongoDB[22] as these databases are easy to scale and are more robust in case of failures. They are schema-less and have faster access times. Thus, NoSQL databases like MongoDB is the next best option to consider.

Our implementation thus provides support to MongoDB database. The database comprises of a single collection under which multiple documents that are focused on each of the metrics are written out in JSON format. These documents can be written to and accessed by MDS in later steps as and when needed.

Apart from the support for MongoDB and MySQL, our tool also provides support for Splunk[56]. Most admins in HPC centers make use of Splunk to quickly find error logs, monitor, analyze and visualize related logs as shown in. Our tool basically writes out files in key-value pairs onto a directory that is being monitored by a Splunk forwarding agent. The forwarding agent monitors the directory for any changes. As and when new files are written out or changes are made to any of the files, they will be propagated to the Splunk Manager. Key-value pairs basically comprise of a key which is the time-stamp at which the logs were collected and the value is the metric collected. Multiple files for each of the metrics are written out to a directory that can be specified in the configuration file that can also be provided as an argument at the startup of DDMT.

Our tool is flexible and provides support for different databases. DDMT can be easily used in any

Lustre deployment for statistics collection.

3.3.3 Polling time

Polling time is the interval between each probe. This is set to a default of 30 seconds and this can be changed in the configuration file based on the requirement.

Polling time is important as in a dynamic HPC system, at any point in time there will be several different jobs running and each of these jobs run different applications and they can each run for a different duration.

The time interval between each of the probes to the different components thus plays an important role in later steps of our implementation. If the polling time is too less, then MDS will be loaded with too many unimportant statistics. On the other hand, if the polling time is too long, then we may lose critical system information. Thus, choosing an ideal polling time is important.

3.3.4 Dedicated plug-ins for each of the metrics

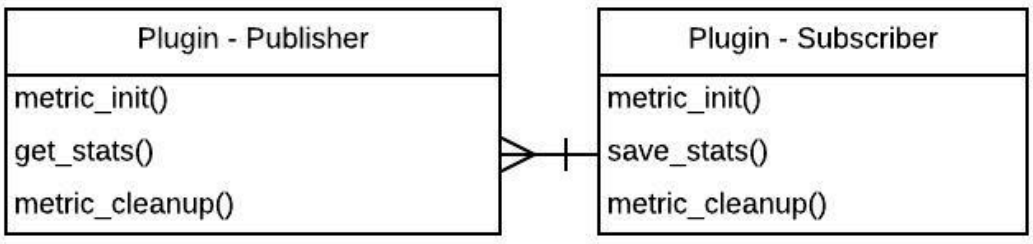

Figure 3.4: Plug-ins for publisher and subscriber.

DDMT has a specific plug-in that is dedicated towards collection of a particular metric [Figure 3.4]. If a specific metric is unnecessary we can easily turn off that plug-in. On the other hand, if we wish to collect a new metric we can do so by adding a new plug-in. Thus, plug-ins help in easy extensibility of the tool.

The plug-ins play complementary roles in a publisher and a subscriber. Plug-ins connect the publisher and subscriber processes to the back-end collected database. Each plug-in collects the statistics from the publisher. These statistics are then forwarded to the subscriber. The subscriber then writes out these statistics in database specific format.

Chapter 4

Characterization of Application I/O

4.1 Overview

Scientific applications exhibit repeating patterns in their requests. These requests generally occur in bursts and we exploit this repeating pattern in our implementation.

4.2 Design

Scientific computing applications have been shown to exhibit distinct patterns [59] as depicted in Figure 4.1. Given the longevity of legacy applications, and our interactions with HPC practitioners, such predictable behavior is expected to be true in emerging applications as well.

Therefore, application behavior can be modeled to help predict future application requests. We capture the repeating pattern in application requests by making use of machine learning techniques. Applications request for certain number of read bytes or write bytes in every snapshot. There could be some instances when either of the requests are not being made.

We try to focus on write requests made by the application. This is because it is at the time of application write that an OST will have to be picked by the MDS and handed over to the client. Thus, our goal of reaching a load balanced state can be achieved only by considering write bytes.

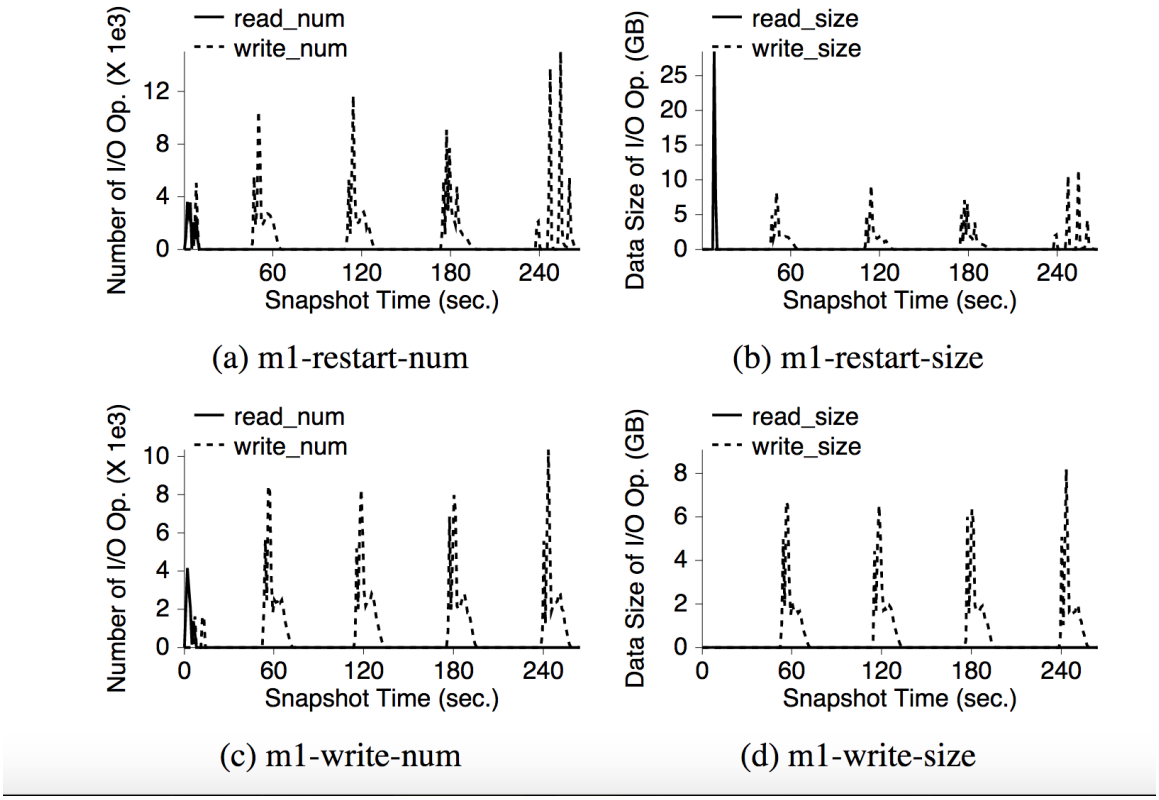

Figure 4.1: I/O requests over Time for a Physics Application(m1) [59]

Read requests on the other hand, can only be serviced from the OST that is currently holding the particular file stripe. Also, since in the later stage, OSS load is being considered, we need not worry about the load on the file system being affected by the read requests.

Another important parameter that needs to be predicted is the number of OSTs that will be requested. This parameter is again very important to predict beforehand. With the knowledge of how many OSTs may be requested in the future an optimal load balancing decision can be made.

4.3 Approach

In order to predict the number of write bytes and the number of OSTs that will be requested by the application, several different approaches were considered. Time-series prediction, Hidden Markov Model and Reinforcement Learning were the different approaches that were tried but the results

from Markov-chain model was more suited to our system.

4.3.1 Markov Chain prediction

Scientific computing applications have been shown to exhibit distinct patterns [59]. Given the longevity of legacy applications, and our interactions with HPC practitioners, such predictable behavior is expected to be true in emerging applications as well. Therefore, application behavior can be modeled to help predict future application requests. We leverage this information by identifying three key predictable properties of high performance computing applications' I/O requests:

- Inter-arrival rate of requests (IRR).
- Number of requested OSTs (stripe count).
- Number of bytes to be written in each request.

We focus on balancing the load during file creation. Thus, we only use the *number of bytes written* in the request. If the request is a read request, the number of bytes written will be zero. We focus on write bytes more than read bytes for load balancing because once the file writes have been distributed, the caching mechanism and buffers can absorb much of the read requests, thus mostly protecting read performance from the load imbalance at the OSTs. Therefore, file read requests are expected to not have a significant impact on the load of the storage system as is the case for write requests.

Markov Chain Model has previously been employed in predicting the spatial pattern of I/O requests in scientific codes [46]. This work tried to model the access pattern of blocks in storage systems in parallel file systems.

We use a Markov Chain Model [50] to capture the three properties (IRR, stripe count, bytes written) of application requests, factor in the current patterns, and predict future requests. The predicted requests along with the current application requests will then be used to drive our load balancer.

Markov chain is a process in which the outcome of an experiment is affected by the outcome of a previous experiment. If a future state in the model depends on the previous m states, it is termed as

a Markov chain of order m. Therefore, more memory can be built into the model by using a higher order Markov model. But, as the order of Markov chain model becomes higher, the estimates of different parameters become less reliable.

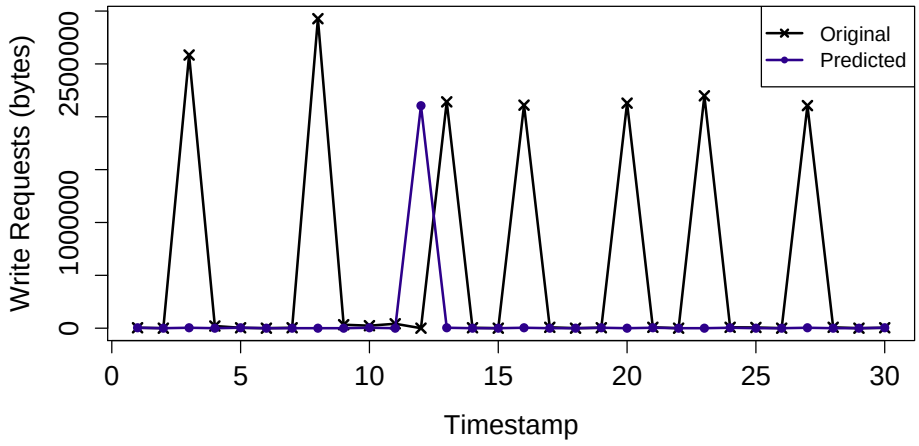

Figure 4.2: Orig. vs. predicted requests (Order = 1)

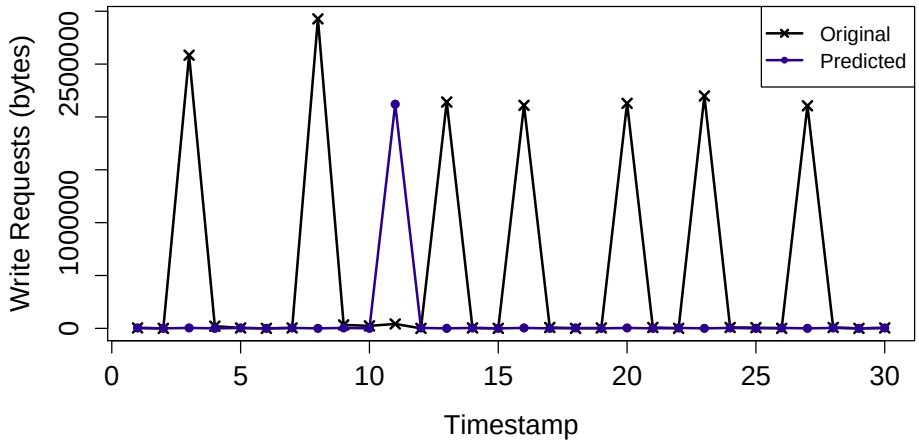

Figure 4.3: Orig. vs. predicted requests (Order = 2).

Order of Markov chain	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Execution time (seconds)	0.01	0.02	0.15	1.0	2.6	3.8	5.6	8.0
Accuracy (%)	65.6	78.7	95.5	96.3	96.8	97.2	97.5	97.8

Table 4.1: Execution time and accuracy % for Markov chain for varying orders.

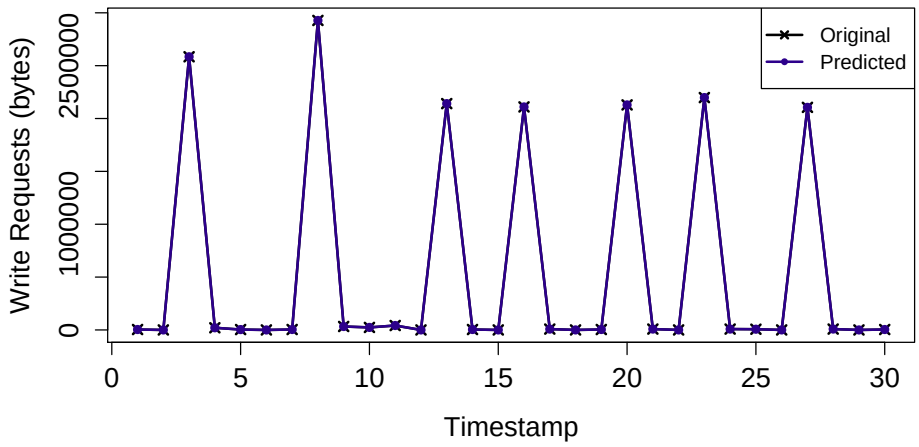

Figure 4.4: Orig. vs. predicted requests (Order = 3).

Figures 4.2, 4.3 and 4.4 show the predictions from Markov chains of varying orders on a collective trace of three different applications running simultaneously on the system. Here again, two of the application traces are from IOR benchmark [41] and HACC IO benchmark [40], while the third application trace is generated from a Gaussian distribution having the mean write bytes and read bytes as 2048 and 1024 bytes, and a variance of 4096 write bytes and 2048 read bytes. The results for orders 1 and 2 show that predicted write bytes are different from the actual bytes requested to be written by the application. We get the best results for order 3. We tested our model for higher orders as well, and the results are shown in Table 4.1. All results are from the average of three runs. The results show that while the increase in accuracy from using a higher order model chain (greater than 3) was minimal, the corresponding execution time became very high. Thus, we do not go beyond fourth order chains in our approach.

States are defined for every component of the application request that is being modeled. For

Range of write bytes	**Mean**
0 - 1000	500.5
1001 - 4000	2500.5
4001 - 16000	10000.5
16001 - 48000	32000.5
48001 - 192000	120000.5
192001 - 768000	480000.5
768001 - 3072000	1920000.5
$>$ 3072000	3072001

Table 4.2: States for write bytes.

example, to lower the error in the prediction of number of bytes to be written, the states defined for the *bytes written* property are mean bytes written. The ranges and their means (as used for the states) are shown in Table 4.2. The process starts in one of the states S_i and moves onto the next state S_j with probability p_{ij}. These probabilities are the transition probabilities and are captured by training the model on the requests observed so far. We utilize dynamic learning, wherein the probabilities are continuously updated as more requests are processed.

For example, when predicting the number of bytes written, every request that arrives is stored as a key with value as the requests that follow. Since, our model gives the best results with a Markov chain of order 3, we store three following requests. We then predict the values based on the observed behavior, completing the process. The Markov chains associated with stripe count and IRR are handled similarly.

The training sequence of requests is a list of any length longer than the order of Markov chain, from which the model draws state information for the chain. Our predictor dynamically determines the length of the training sequence. The dynamic approach is needed to accommodate the fact that the request size and pattern will vary for different applications as well as over time. We capture such changes using mean square error computation. To this end, we find the mean square error for different lengths of training sequence (e.g., 2, 4, etc.) to find the length that gives the least mean square error and the model then uses that length for training. We limit the length to 16 to

keep the computational overhead of our approach in check. Our Markov model is also application independent, that is, we do not need to build new models for different applications. The same model can learn different applications behavior and assign them different identifiers. The model can then differentiate the applications based on their assigned identifiers and predict the appropriate set of application-specific requests.

Threshold determination for different workloads can be done in many different ways. The work on distributed Load Balancing in Key-Value Networked Caches determines load by factoring in the maximum load and average load [35]. Load on the server is also determined by considering the global logs [18] and in some other systems load is determined by factoring in the number of CPUs [4].

In our implementation, we determine the threshold by initially setting the threshold to be the current load, we then update the threshold to the mean of the loads seen so far. After a 30 second interval, a daemon process checks if the load on the MDS is within the average load seen so far. If it is, Markov Chain model is run and the old values are purged in the buffer. Till the next time the model is run, we make use of the same set of predicted requests. We adopt this approach because MDS load is dependent on the inter-arrival rate of requests as well as on the rate at which files are accessed or updated. Adopting this approach ensures that we are always within safe-bounds while running the Markov Chain model. This technique also ensures that the Markov Chain will be run after some amount of time even in conditions where the load on MDS varies drastically.

Chapter 5

Load Balancing Model

5.1 Overview

We make use of the current usage statistics and the predicted application requests in order to come up with the load balancing algorithm. The load balancing algorithm tries to even out the load on the OSS and the OST layer.

5.2 Design

We can break down the algorithm as follows. We have as input,

- the current request,
- the next set of predicted requests,
- the current load on OSS,
- and the usage of OST.

We need to select as many OSTs as requested by the client. During the selection of these OSTs we need to ensure that the selected OST has enough space to write the requested number of bytes and

at the same time we need to ensure that the OSTs that are selected do not overload the OSS.

We also need to ensure that all the OSTs get used over time and if an OSS or OST is already busy servicing another request then we should try and find the next best available OSS or OST.

5.3 Approach

5.3.1 Distributed Dynamic Load Balancing Algorithm (DDLB)

We try to make use of the existing framework, DDMT, in order to devise our load balancing algorithm.

At each layer of data collection, we turn on the plug-ins of DDMT in such a way that we collect only the most important metrics that are needed for load balancing. This way we filter out the important metrics and ensure that only these are transmitted across the network [53].

We initially start off at the OST layer. Here the most important metric for us is the amount of space that is available (KB_avail).

We then move up to the OSS layer. In order to load balance effectively we come up with a cumulative metric for load that takes into account both the memory and CPU usage. We assign both the parameters equal weight and have a single combined metric for load.

$Load = (\alpha(\%\ of\ CPU\ Utilization) + \beta(\%\ of\ Memory\ Utilization))$.

α and β are the weights associated with CPU and memory utilization, respectively. For our purpose, we give both and a value of 0.5. This could be changed later if the requirement or the behavior of the system changes.

At MDS, we receive two metrics KB_avail per OST and Load of each OSS.

Markov chain model for prediction is run at MDS as explained earlier. This model will predict the next set of requests. This knowledge is made use of in ensuring that entire system stabilizes towards a load-balanced state given the next set of predicted requests. We can therefore, ignore the accuracy towards the tail of Markov chain prediction as our intention is to have only the initial requests

to be accurate and the remaining predicted requests to only be representative of the application characteristic.

We thus, have all the important metrics at MDS.

- KB_Avail on each OST
- Load on each OSS
- Current request
- Next set of predicted requests

Our algorithm should not just look at each of these metrics independently and optimize on them but it should also look at these metrics holistically and take into account the inter-dependence of each of the metric on one another.

If we make a decision based on the KB_avail on the OST alone then we may end up overloading the OSS.

If we just try to satisfy the current request, the system on the whole may not reach a load balanced state.

We thus, come up with the below approach in order load balance the whole system. The below formulation of the algorithm makes use of greedy approach to determine a mapping between requests and OSTs. Similar approaches have previously been employed in Domain Redirection algorithms for Load sharing in Distributed Systems [19].

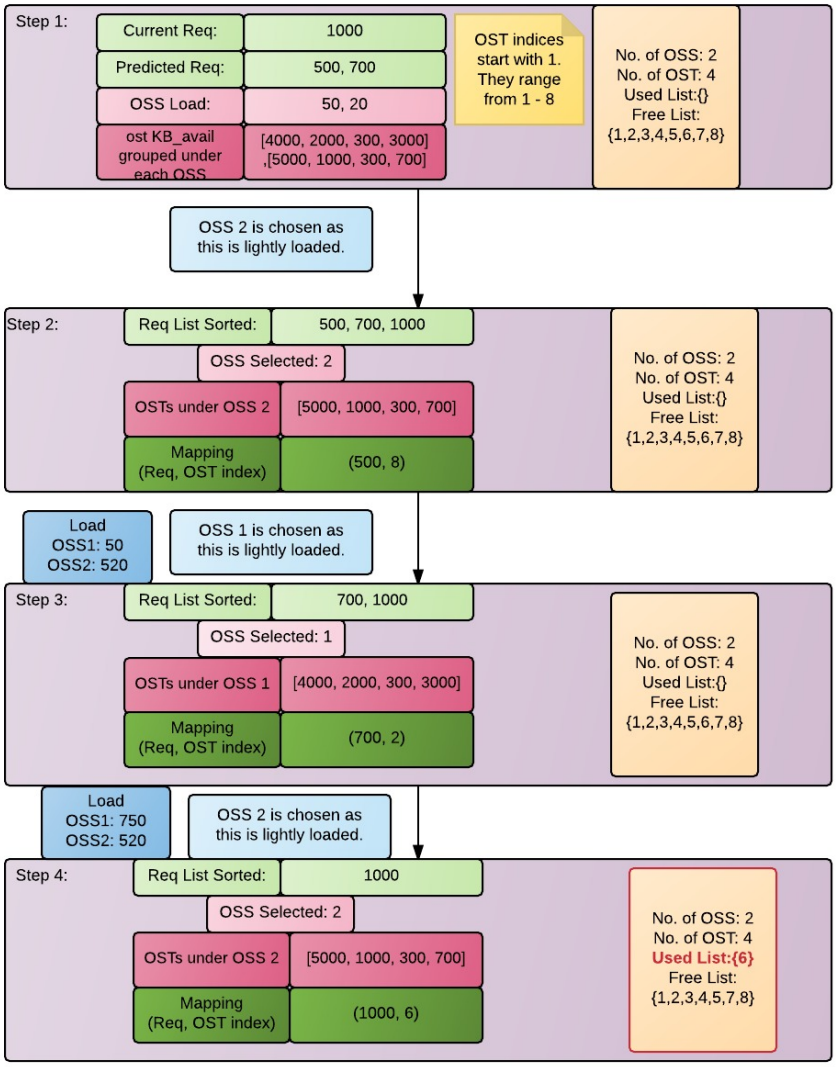

Figure 5.1: One iteration of Load Balancing Algorithm

Our intention is to bring the whole system to a load balanced state. The below algorithm is illustrated with an example input in Figure 5.1. The algorithm depicts one iteration in the algorithm.

We need to find a mapping between the requests(both current and predicted) and OSTs such that the whole system is balanced.

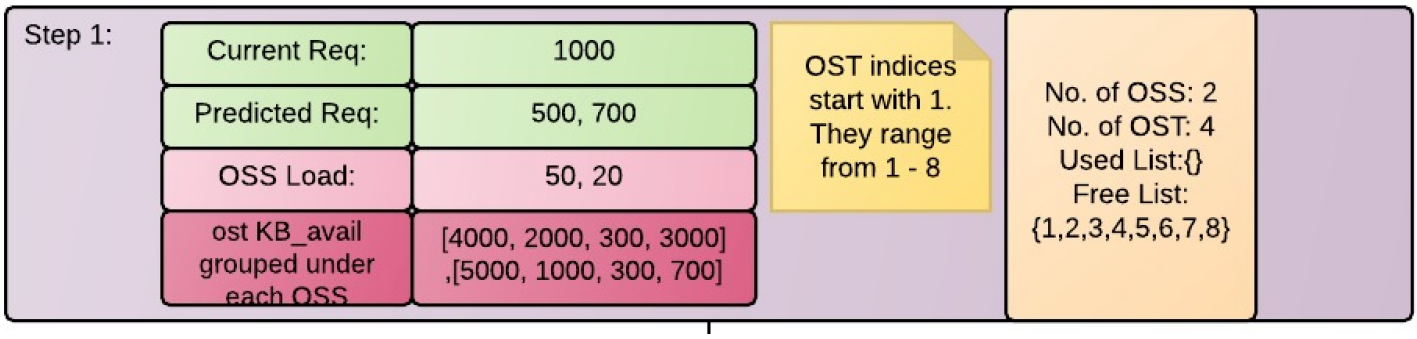

Figure 5.2: Step 1: Initial state of the system. Current Request is the number of write bytes being requested at a particular instant. Predicted request is the number of write bytes that could appear next. OSS Load is the current load on the OSS. This load in the example will be updated based on the number of bytes being written through the server. OST KB Avail is the amount of space in bytes available on each OST. In this example, there are two OSSs and each OSS has four OSTs. There are thus, two lists of OSTs.

In order to achieve this, we order the requests in such a way that we start by satisfying the lowest possible request. And we move further from there to find the next best match for the next highest request and so on.

We then move forward to choosing an OSS that is lightly loaded. This way we ensure that the same OSS does not end up satisfying all the client requests and at the same time this best utilizes the parallel architecture of the file system as the load now gets uniformly distributed across all OSSs.

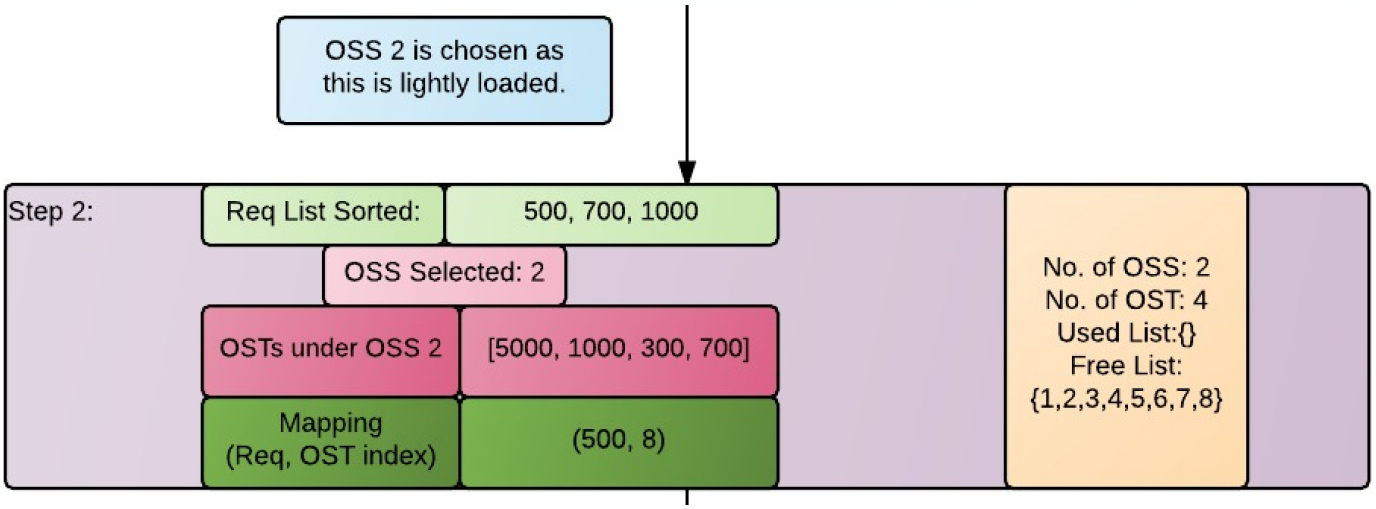

Figure 5.3: Step 2: OSS 2 with the least load is selected. A mapping for the request with least bytes, i.e, 500 bytes is found. An optimal OST under OSS 2 is chosen. The best match is the OST with 700 bytes. This OST has an index 8. Indices of OSTs start from 1. Since this is the fourth OST under second OSS it's index is 8.

Once we have chosen the lightly loaded OSS, we select the OST with the best available space to satisfy the request under consideration.

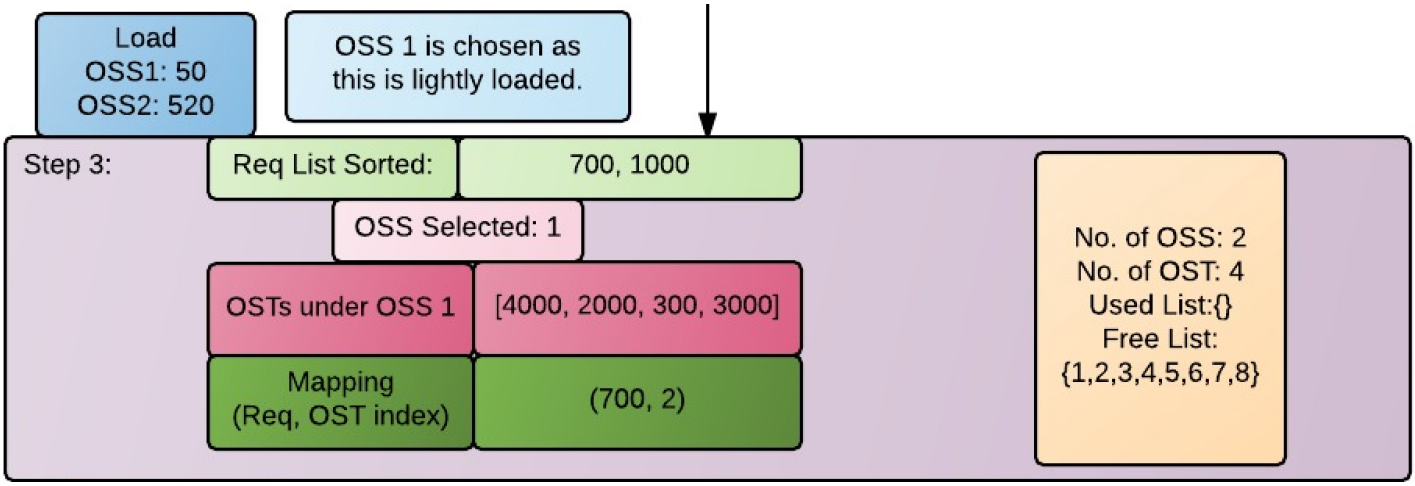

Figure 5.4: Step 3: OSS 1 with the least load is selected. A mapping for the request with least bytes, i.e, 700 bytes is found. An optimal OST under OSS 1 is chosen. The best match is the OST with 2000 bytes. This OST has an index 2.

As mentioned before, this helps to account for the worst case request of a very high request coming in future. If a request higher than predicted comes in, then there is room even for that.

If none of the OSTs can satisfy the request under consideration, then we consider the next lightly loaded OSS and try to satisfy the request.

Once the set of OSTs that satisfies the current request is found, then these OSTs are moved to used state.

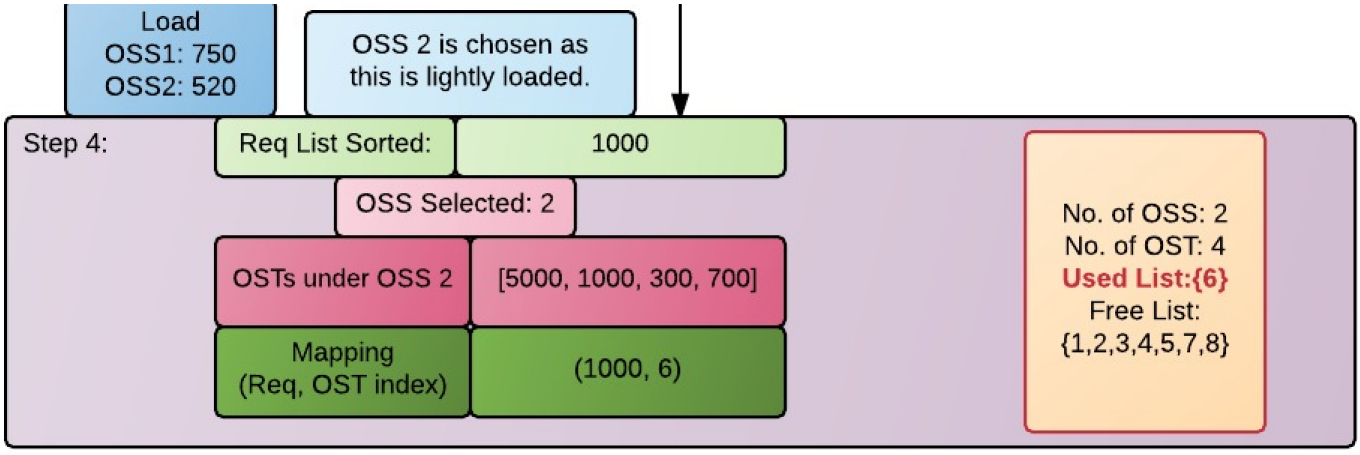

Figure 5.5: Step 4: OSS 2 with the least load is selected. A mapping for the request with least bytes, i.e, 1000 bytes is found. An optimal OST under OSS 2 is chosen. The best match is the OST with 1000 bytes. This OST has an index 6. Since at the end of this iteration, both current and predicted requests have been assigned a mapping. MDS returns the OST corresponding to the current request, i.e., OST 6 and moves this OST to the used state. This whole process is repeated again for the newer requests as and when they come.

If on the other hand, even after probing through all the OSSs we do not find a OST that can satisfy the request, we try probing through the OSTs that are in the used states and let this request wait on that OST.

In the worst case scenario of no OST satisfying the request, we have an error handling function that will log this message so that the system administrators can take necessary action.

We maintain used list and free list in order to force our algorithm to pick those OSTs that have not

been picked before and also to account for the variability in the bandwidth of different OSTs.

5.3.2 Max-flow algorithm and Bipartite matching for load balancing

DDLB can be intuitively mapped on to max-flow algorithm and bipartite matching algorithm.

For this purpose, we use a maximum bipartite matching algorithm [34]. The algorithm creates a matching of the maximum number of edges in a bipartite graph, with edges chosen in such a way that no two edges share an endpoint. We also consider the maximum flow problem [29], which is solved for a graph that represents a flow network in which all edges have a capacity associated with them. The problem aims to find the maximum flow from source to sink. We combine the maximum bipartite matching and maximum flow algorithms to design a Dynamic Maximum Flow Algorithm (DMFA) for the purpose of load balancing.

In the default bipartite matching algorithm, the edges have a capacity of one. In our DMFA, the capacities are modified to stripe count. This enables a request to be mapped onto more than one OST. Figure 5.6 shows the formation of our graph.

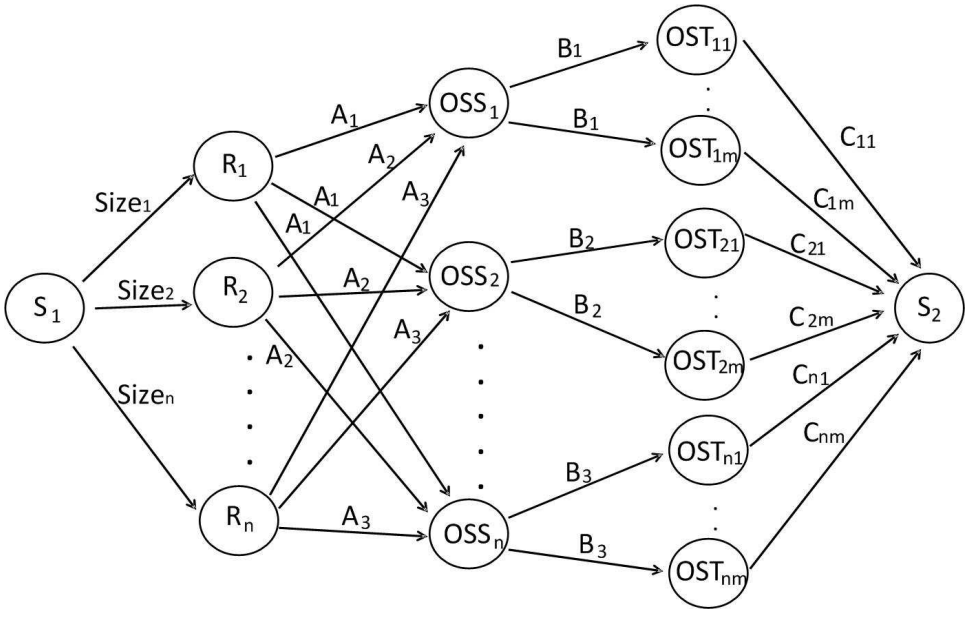

Figure 5.6: Graph used in our dynamic maximum flow algorithm.

In the graph, S_1 and S_2 are the source and sink nodes, respectively. $R_1, R_2, ..., R_n$ are the combination of current and predicted application requests. The weight of the edges between source node and application requests is the size of the request (in bytes). We assign a priority to each request node, with current requests having a higher priority than that of predicted requests. The next set of nodes consists of OSSs. A request is connected to all the OSSs. The capacity of the edge between a request (R_i) and all OSSs is shown as A_i, where $A_i = WriteBytes_i/StripeCount_i$. Therefore, every edge from a request contains the bytes to be written on one OST. Each OSS is connected to its associated set of OSTs. The capacity of the edge from an OSS OSS_i to its OSTs is given as B_i, where $B_i = (\alpha + \beta)/(\alpha(CPU\%_i) + \beta(Memory\%_i))$. α and β are the weights associated with CPU and memory utilization, respectively. We assign both values as 0.5, thus giving equal weights to both (the weights can be modified to accommodate applications where one or the other resource has higher constraints). We assign the capacity of the edge to be an inverse of the load associated with an OSS. This is because, in a maximum flow problem, the lower the load on an OSS, the higher should be the flow. Finally all OSTs are connected to the sink (S_2), which is responsible for maintaining a load balanced setup. The capacity of the edge from OST_{ij} (OST_i of OSS_j) is shown as C_{ij}, where $C_{ij} = KBavailable_i - \mathrm{Re}questedWriteBytes$. For the initial state of the graph, the requested write bytes are zero. Therefore, if an OST is selected for a particular request, the capacity of the edge between that OST and sink node (S_2) is updated to the difference between the capacity available on that OST and the requested bytes to be written.

Finally, we use the Ford-Fulkerson Algorithm [29] to solve the dynamic maximum flow problem. Residual graph in the flow network indicates additional possible flow in the network. Augmenting paths are found in the residual graph, which adds to the flow in the graph. We use priority first search to find augmenting paths. Requests are assigned priorities because the actual application requests should have higher priority at finding correct OSTs than the predicted (error-bound) requests. The augmenting path that is responsible for giving the maximum flow is selected first and the request is mapped onto the OST on that path. The next maximum path is then selected and this continues until all the requests have been assigned to the OSTs. This completes the current allocation round, while ensuring even load distribution across the OSTs.

Chapter 6

Evaluation

We evaluate the efficacy of our approach using a Lustre simulator. In the following, we first describe our simulator and experimentation methodology, then compare our DMFA load balancing with the default Lustre OST allocation approach.

6.1 Methodology

6.1.1 Lustre Deployment Simulator

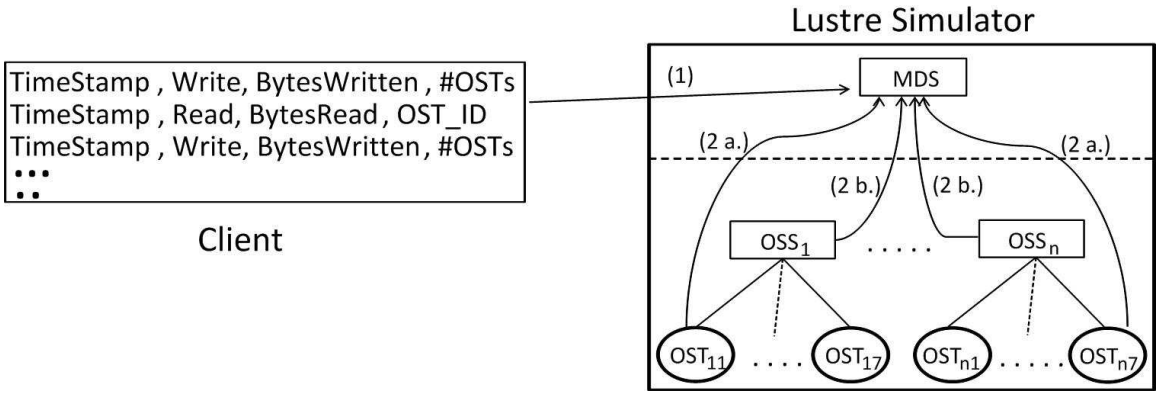

Figure 6.1: Lustre simulator architecture.

Sun Microsystems, Inc, developed a simulator for Lustre in 2009 [6]. A lot of changes has gone into Lustre since then and this has thus made it necessary for us to build a custom simulator for our work.

We have developed a discrete-time simulator as shown in Figure 6.1 to test our approach at scale. The simulator has four key components closely mirroring those of Lustre's, namely OST, OSS, MDT, and MDS, which implement the various Lustre operations and enable us to collect data about the system behavior. The MDS is also equipped with multiple strategies for OST selection, such as round-robin, random, and DMFA. We have implemented a wrapper component that enables communication between our various simulator components. The wrapper is responsible for processing the input, managing the MDS, OST, and OSS communication and data exchange, and driving the simulation. The application traces collected from client side are modeled as clients in the simulator. Section 6.1.2 provides the application details.

In our simulations, all OSTs are assigned the same bandwidth at the start. The simulator also takes into account the number of applications using a particular OST at a given time-stamp and calculates the read and write bandwidth accordingly. We assume that OSTs have equal read and write bandwidth. The parameters, number of OSSes, number of OSTs under each OSS, and the ratio of read and write bandwidth can be provided as inputs to the simulator.

6.1.2 Workloads

To drive our simulations, we collect application traces at the client side. These application traces contain two kinds of entries: (a) write entries, which have the time-stamp, the number of bytes to be written, and the number of OSTs to be selected (i.e. the stripe count); and (b) read entry, which has the time-stamp, number of bytes to be read and the OST ID from which the bytes have to be read.

To model the behavior of a real Lustre deployment, we run and capture a trace of 3 simultaneously running applications on a local Lustre deployment. Two of the applications used are: the HACC I/O benchmark [40] [55] that measures the I/O performance of the system for the simulation of Hardware Accelerated Cosmology Code (HACC), and the IOR benchmark [41] that is used for

testing the performance of parallel file systems. The third application trace is generated from a Gaussian distribution having the mean write bytes and read bytes as 2048 and 1024 bytes respectively, and a variance of 4096 write bytes and 2048 read bytes. For our tests, we simulate the behavior of one MDS, eight OSSs with four OSTs per OSS, for a total of 32 OSTs.

6.2 Comparison of Load Balancing Approaches

We compare our DMFA based OST load balancing with the standard Lustre round-robin approach, as well as two additional load balancing approaches: (a) random allocation where OSTs are selected at random; and (b) a heuristic based model. Our heuristic based model sorts all the OSTs based on the ratio of current bytes to store to available space and then picks the OST with the least ratio. The goal is to allocate an incoming request to the least loaded OST.

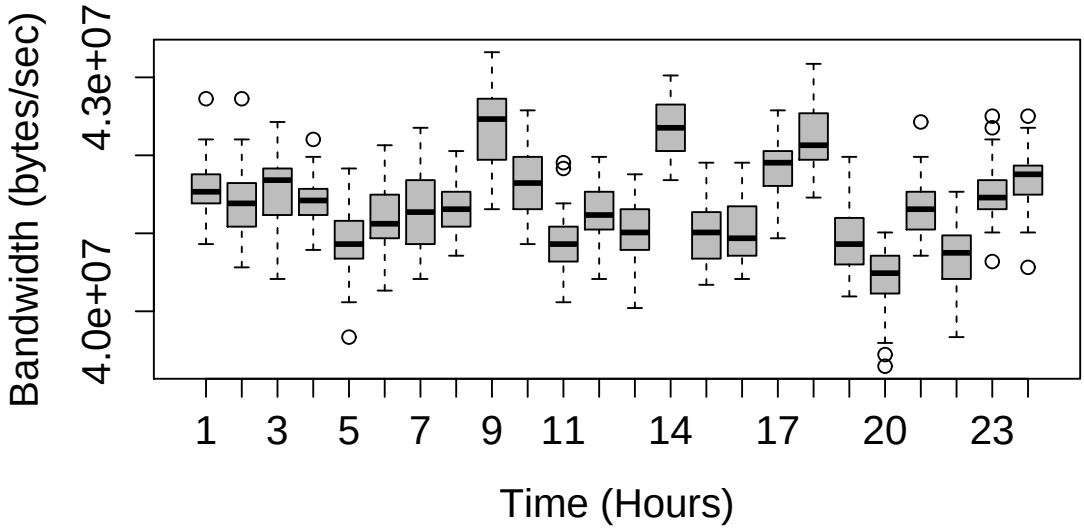

Figure 6.2: Mean bandwidth of all OSTs over time under Random.

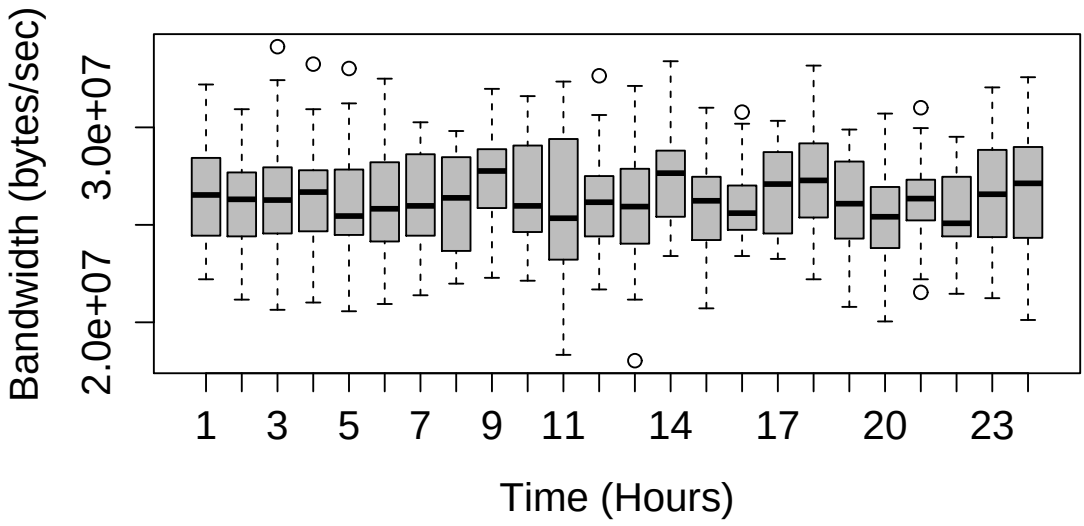

Figure 6.3: Mean bandwidth of all OSTs over time under Round Robin.

Figure 6.2 shows the mean bandwidth of all OSTs per hour over a period of 24 hours for random OST allocation. We studied random allocation to make the case for a more informed allocation, and also to quantify the impact of choosing a wrong allocation approach. For every hour, the figure shows the mean bandwidth of all OSTs as a box plot. The observed huge variations in the mean bandwidth shows that random allocation performs worse than even Lustre's default round-robin approach (Figure 6.3).

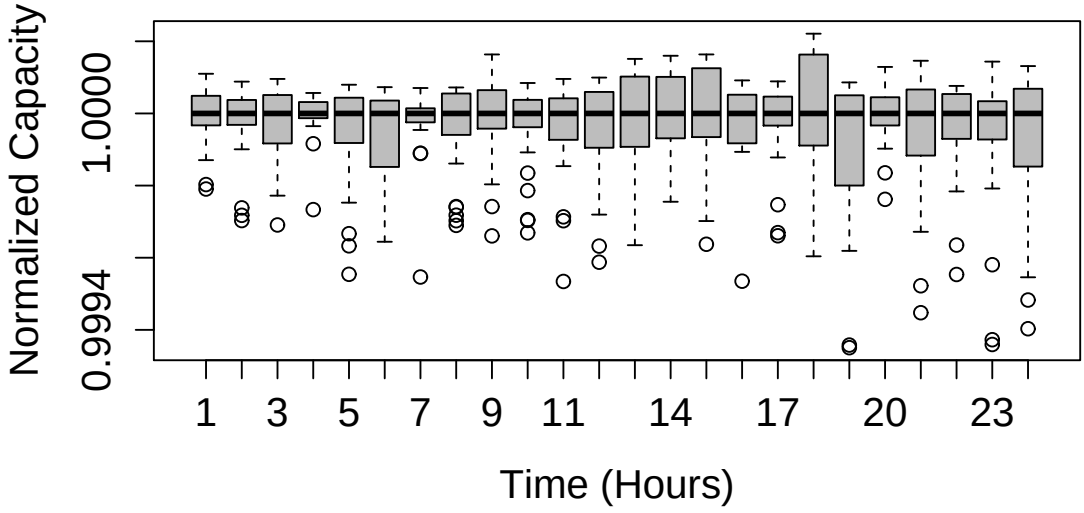

Figure 6.4: Capacity of all OSTs over time under heuristic based model.

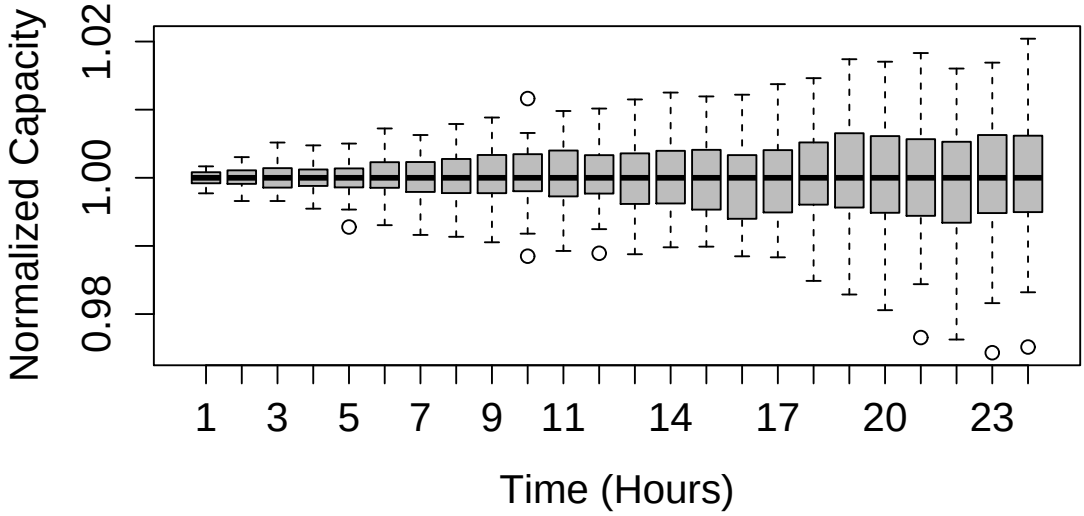

Figure 6.5: Capacity of all OSTs over time under Round Robin Model.

Our objective is to load balance OSTs such that every OST is at an almost similar state (in terms

of number of bytes available, and load) under various file allocations. For this purpose, we also studied the heuristic based model to determine its effectiveness in achieving better load balancing. Figure 6.4 shows the normalized capacity of all OSTs per hour for 24 hours using the heuristic model. Since capacity of OSTs continuously decreases over time, we use normalized capacity where for every hour, the capacities of all OSTs are divided by the median capacity. The box-plot shows that the heuristic model does not perform well because the inter-quartile ranges (IQR) for some of the hours are much greater than the IQRs in case of default round-robin allocation (Figure 6.5). Also, the increasing number of outliers shows that the heuristic based model is not suitable for a load-balanced allocation of OSTs.

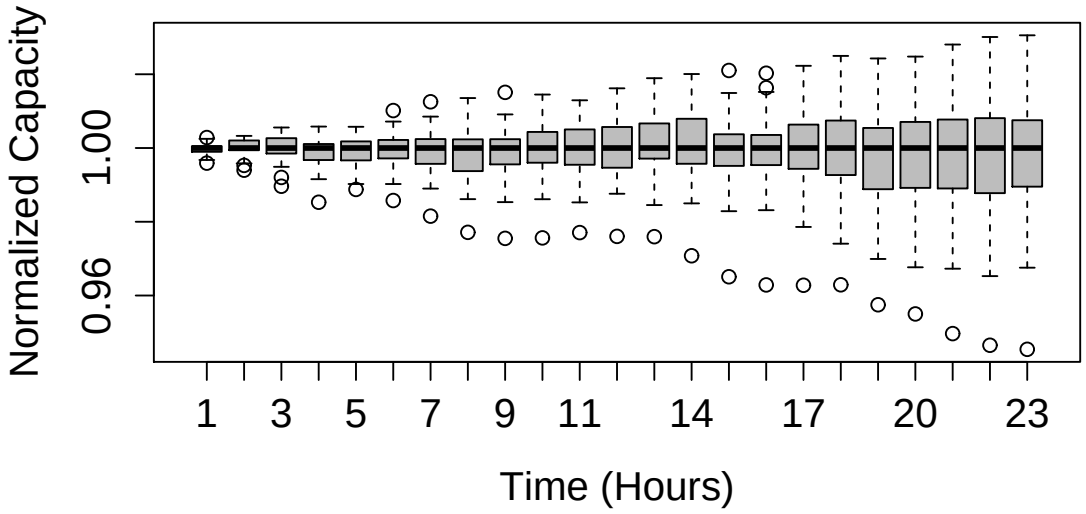

Figure 6.6: Capacity of all OSTs over time under Maximum Flow.

In the next set of experiments, we compare our DMFA approach with Lustre's default round-robin OST allocation. First, we study the load of OSTs under the two approaches. Figure 6.6 shows the box-plot for normalized capacity vs. time for 23 hours under our approach. When comparing this with the round robin approach (Figure 6.5), we see that the inter-quartile ranges for DMFA are less wider than those for round-robin allocation. This shows that at any given time, DMFA is better able to balance load across OSTs.

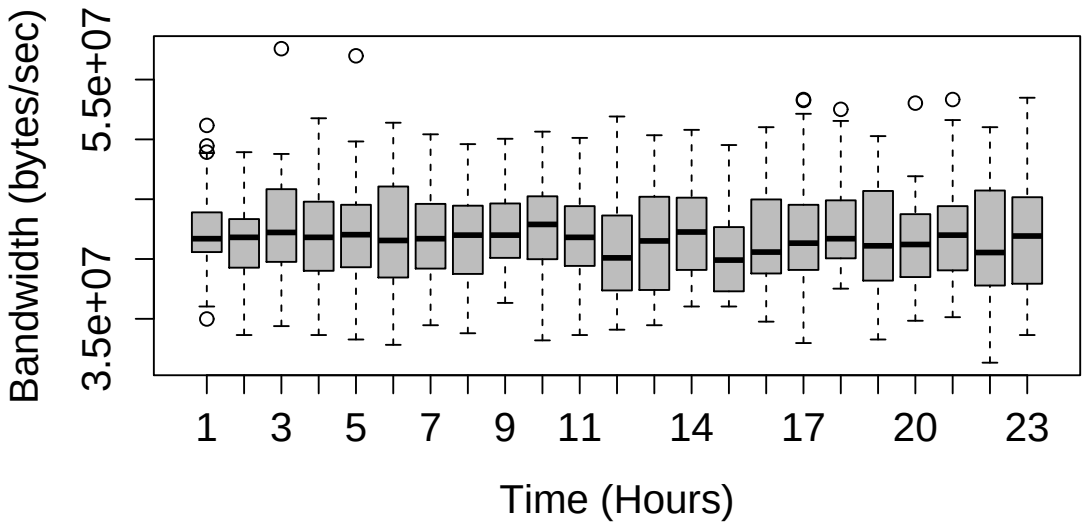

Figure 6.7: Mean bandwidth of all OSTs over time under Maximum Flow.

Next, we test the mean bandwidth of all OSTs over time to determine whether the DMFA based load balancing of OSTs performs better than round-robin allocation of OSTs (Figure 6.3). Figure 6.7 shows a box-plot of the mean bandwidth using maximum flow algorithm for allocation. We see that the median for mean bandwidth of all OSTs across time is consistent over time in the case of DMFA approach. This is in contrast to the round robin approach, where the medians vary for every hour. This shows that our algorithm supports better and more consistent OST bandwidth than the default allocation in Lustre.

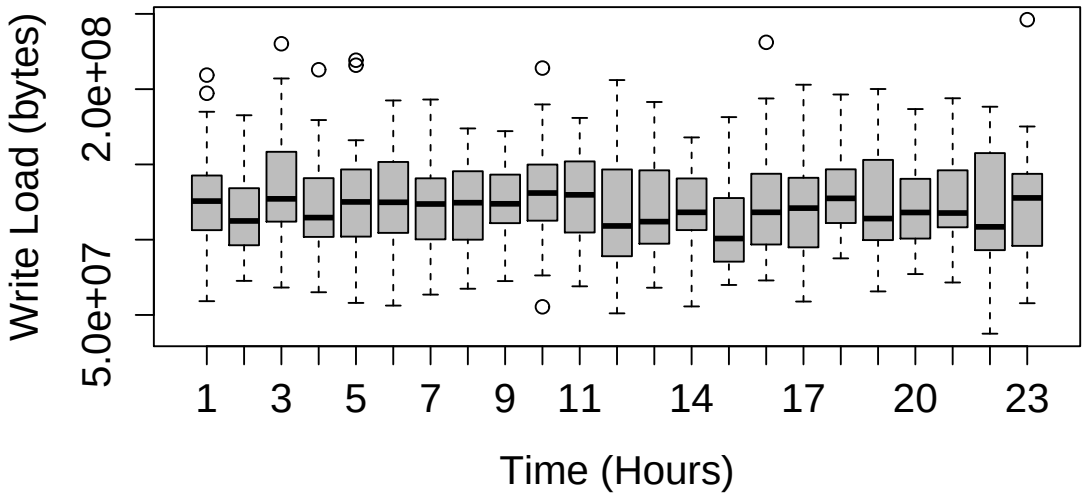

Figure 6.8: Mean write load of all OSTs over time under Maximum Flow.

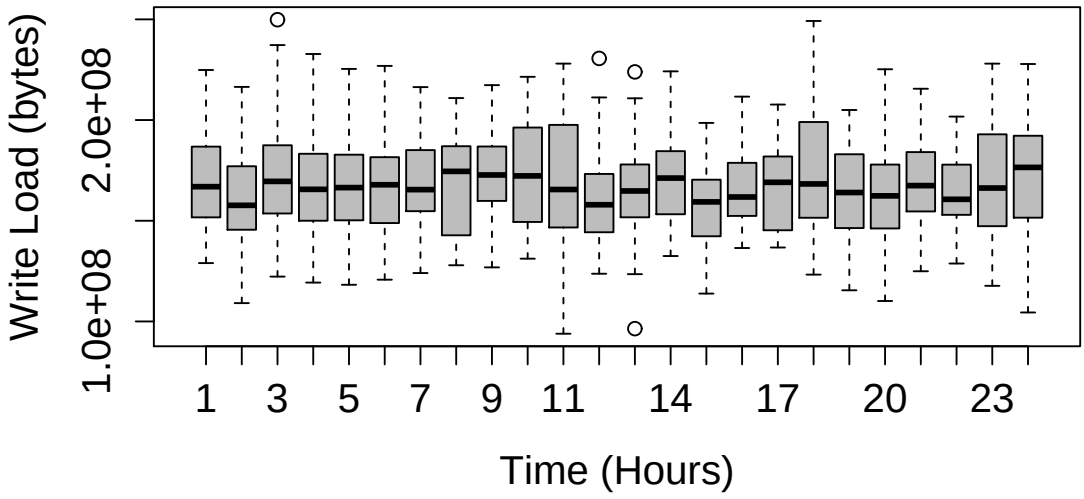

Figure 6.9: Mean write load of all OSTs over time under Round Robin Model.

Moreover, the better and more consistent bandwidth of OSTs in case of DMFA directly results in a

more uniform write load for OSTs over time as shown in Figure 6.8, compared to the round-robin approach as seen in Figure 6.9.

6.3 Scalability Study

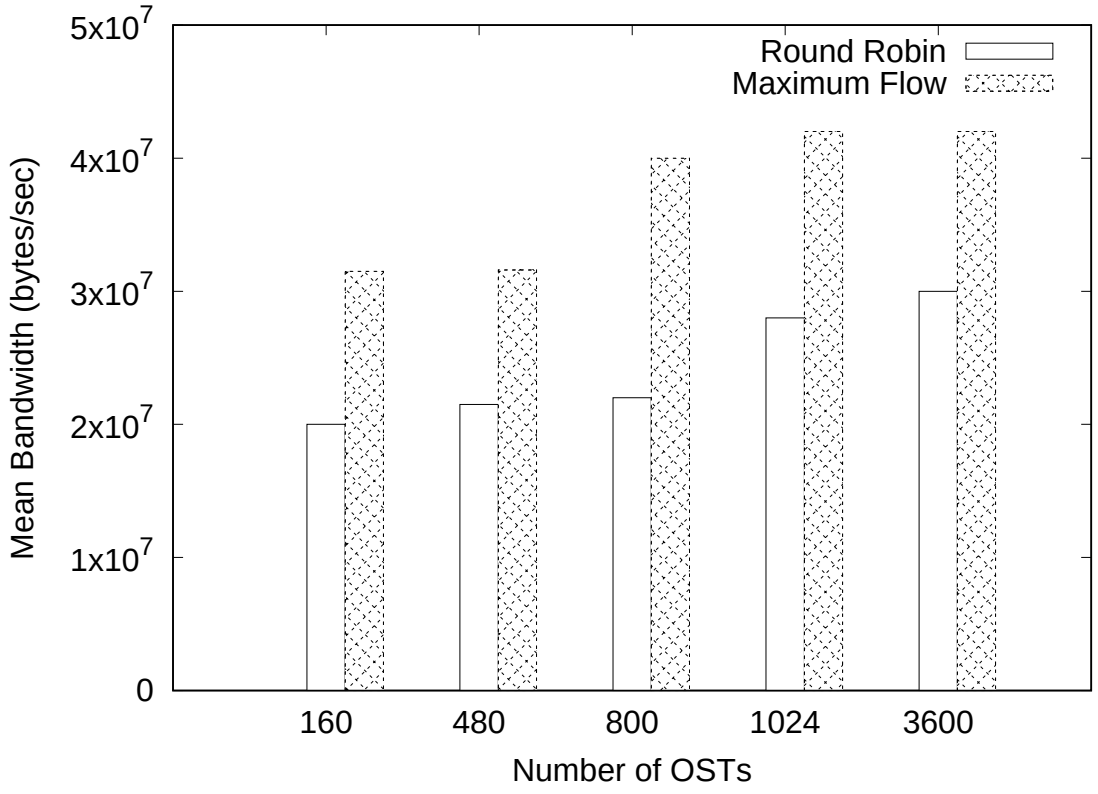

Figure 6.10: Performance with increasing number of OSTs: 160 OSTs (20 OSS), 480 OSTs (60 OSS), 800 OSTs (100 OSS), 1024 OSTs (128 OSS) and 3600 OSTs (450 OSS).

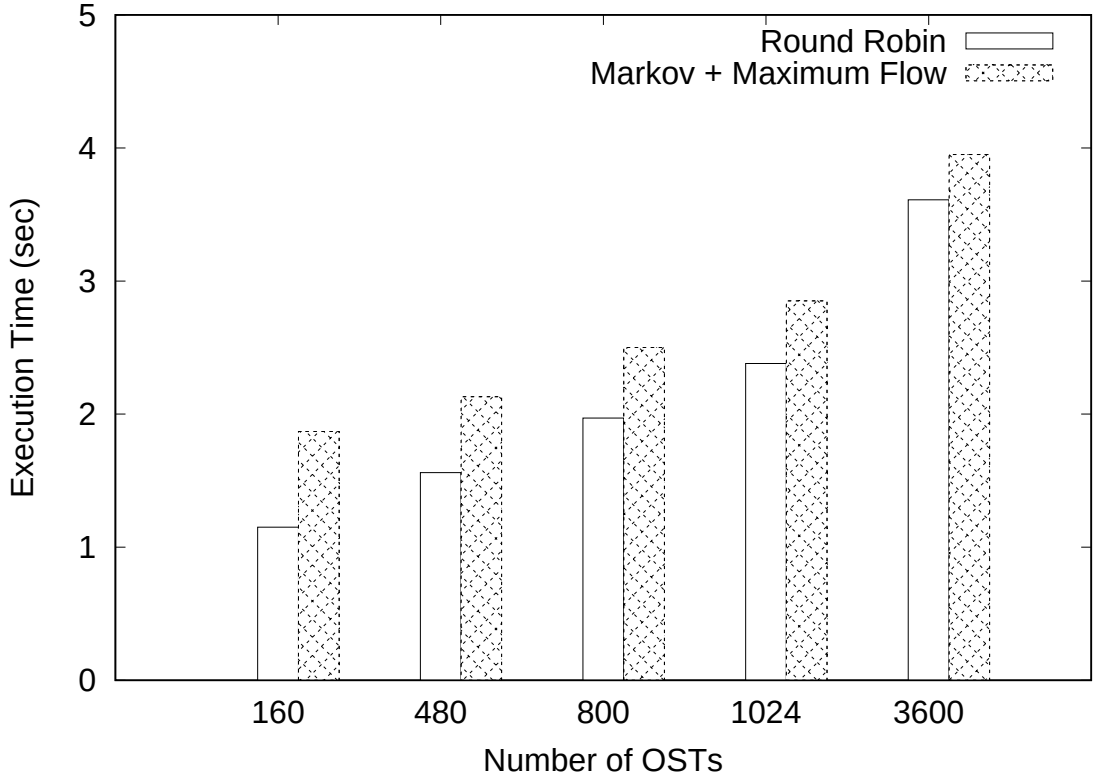

Figure 6.11: Execution time with increasing number of OSTs.

In our next experiment, we test how our approach will work with higher number of storage targets. For this purpose, we use a setup with an increasing number of OSTs from 160 to 3600. Figure 6.10 shows the overall mean I/O bandwidth for Lustre's default round-robin approach as well as our dynamic maximum flow algorithm. As the number of OSTs increases, the mean bandwidth also increases for both the algorithms. Our algorithm provides better performance than round-robin solution even for higher number of OSTs. As seen from the figure, DMFA allocation is able to achieve up to 54.5% (under 800 OSTs) performance improvement compared to the default round-robin approach.

The overall execution time for load balancing using both round-robin approach as well as our algorithm (Markov model along with DMFA) is shown in Figure 6.11. Round-robin takes less time than our approach to allocate OSTs to incoming requests, but the difference between the two execution time keeps on decreasing as we increase the number of OSTs. Even for fewer number

of OSTs, the gain in the overall performance as shown in Figure 6.10 is much higher than the gain in execution time.

These experiments show that DMFA provides better load balanced allocation of OSTs with improved performance compared to Lustre's default round-robin allocation. Also, the performance of our algorithm does not degrade even with very large number of OSTs. Moreover, with higher number of OSTs, the execution time for our approach to allocate OSTs to requests is similar to that of Lustre's default round-robin approach. This is seen in Figure 6.10, where even for higher number of OSTs, DMFA performs better than the standard approach.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

We have presented a load balancing approach for extreme-scale distributed storage systems, such as Lustre, where we enable the system to have a global view of the hierarchical structure and thus make more informed and load-balanced resource allocation decisions.

We have approached the solution by building sub-components of the algorithm. The first necessary sub-component that was designed was the Distributed Data Monitoring Tool. This tool was responsible for collecting all the important metrics from the hierarchical file system and bringing it up to the MDS.

We then predicted the applications' behavior by employing a Markov Chain Model. This model was run at MDS and the model was constantly updated based on the MSE that was observed.

We made use of both these components in order to build a distributed dynamic load balancing model. This model was evaluated on a custom-built simulator against the Round-Robin model. Our algorithm performed better than the extant solution and was able to even out the load both on the OSTs and the OSSs.

7.2 Future Work

1. Distributed Data Monitoring Tool has been implemented in two approaches: Socket Programming based approach and Message Broker based approach. The strengths and weaknesses of each of these approaches has been discussed in detail in the thesis. However, one aspect that still needs to be looked into is the collection of redundant metrics. The tool has been designed to probe the Lustre Components at frequent intervals as defined in the configuration file. There is no mechanism to check if the data that is being collected is the same as that which was collected earlier. In some cases, the storage overhead can be as high as 80%. [13].

2. We can optimize on the exchange of data between the publishers and subscribers by making use of Protocol buffers[57]. This way we can serialize the data and make transfers efficiently. Protocol buffers are compact and fast and can thus enable in faster and more efficient transfer of the collected metrics.

3. Our approach of bringing up all the metrics to the MDS and performing various computations related to prediction of applications' behavior and load balancing algorithm at MDS may prove to be a bottleneck in the future. A distributed approach can be considered.

   Below are some areas that could be explored:

   - Can we run machine learning algorithm in a distributed fashion across lightly loaded OSSs?
   - Can we change the routing algorithm at the LNET layer in order to redirect I/O to different routes based on the current traffic conditions on the inter-connects? The idea here is that initially the client is given a list of OSTs by the MDS. The client tries to write to that particular OST. At the time of object creation, LNET notices that the path to the OST through the OSS is congested and routes the packets to a different OST. Once the file is created on a particular OST, MDS is automatically updated by the OSS. This way all we have to do is to just modify the LNET layer and the remaining layers would continue working like before.

- Can we have a random offset generator for different applications such that the load gets uniformly distributed on different OSTs? This generator needs to monitor only the current usage of the file system and it should be able to make a decision as to which OST offset can be returned to an application in order to even out the load across all OSTs.

4. Although alternatives to Markov Chain were considered in our study. There are still many more approaches that we could experiment with in order to make accurate predictions. We can consider the time-series based neural network approach [15]. We can vary the number of layers and try different activation functions and conduct a detailed study of prediction, based on this approach.

5. ARIMA model [63] was evaluated in our study. This model takes too much computation time and is unable to predict accurately. This is because, this model does not work well with the intermittent series that we are presented with. In order to overcome this we can employ EM algorithm in conjunction with Kalman filter and smooth out the distribution [52]. We could also explore the on-line versions of the time series prediction algorithms that make use of kernel based approaches. We are presented with a large number of inter-dependent metrics and the relationship between each of these metrics is not known. Thus, a kernel-based normalized LMS algorithm may prove to be useful [49].

Bibliography

[1] Cray XT and Cray XE System Overview. https://www.nersc.gov/assets/NUG-Meetings/NERSCSystemOverview.pdf.

[2] Introduction to Markov Chain: simplified! https://www.analyticsvidhya.com/blog/2014/07/markov-chain-simplified/.

[3] K-server problem. https://en.wikipedia.org/wiki/K-server_problem.

[4] Load Monitoring in Linux Servers. https://crybit.com/load-monitoring-in-linux-servers/.

[5] Markov Chains: Explained Visually. http://setosa.io/ev/markov-chains/.

[6] Server Side Request Scheduler. https://bugzilla.lustre.org/show_bug.cgi?id=13634.

[7] Carl Albing, Norm Troullier, Stephen Whalen, Ryan Olson, and J Glensk. Topology, bandwidth and performance: A new approach in linear orderings for application placement in a 3d torus. *Proc Cray User Group (CUG)*, 2011.

[8] Ali Anwar, Andrzej Kochut Anca Sailer, Charles O. Schulz, Alla Segal, and Ali R. Butt. Scalable metering for an affordable it cloud service management. In *Proceedings of the IEEE International Conference on Cloud Engineering*, IC2E '15, Tempe, AZ, March 2015.

[9] Ali Anwar, Yue Cheng, Aayush Gupta, and Ali R. Butt. Taming the cloud object storage with mos. In *Proceedings of the 10th Parallel Data Storage Workshop*, PDSW '15, pages 7–12, New York, NY, USA, 2015. ACM.

[10] Ali Anwar, Yue Cheng, Aayush Gupta, and Ali R Butt. Mos: Workload-aware elasticity for cloud object stores. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, pages 177–188. ACM, 2016.

[11] Ali Anwar, Yue Cheng, Hai Huang, and Ali R Butt. Clusteron: Building highly configurable and reusable clustered data services using simple data nodes. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*. USENIX Association, 2016.

[12] Ali Anwar, K. R. Krish, and Ali R. Butt. On the Use of Microservers in Supporting Hadoop Applications. In *Cluster Computing (CLUSTER), 2014 IEEE International Conference on*, Madrid, Spain, Sep 2014.

[13] Ali Anwar, Anca Sailer, Andrzej Kochut, and Ali R Butt. Anatomy of cloud monitoring and metering: A case study and open problems. In *Proceedings of the 6th Asia-Pacific Workshop on Systems*, page 6. ACM, 2015.

[14] Ali Anwar, Anca Sailer, Andrzej Kochut, Charles O Schulz, Alla Segal, and Ali R Butt. Cost-aware cloud metering with scalable service management infrastructure. In *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on*, pages 285–292. IEEE, 2015.

[15] E Michael Azoff. *Neural network time series forecasting of financial markets*. John Wiley & Sons, Inc., 1994.

[16] Eric B Boyer, Matthew C Broomfield, and Terrell A Perrotti. Glusterfs one storage server to rule them all. Technical report, Los Alamos National Laboratory (LANL), 2012.

[17] Peter J Braam and Rumi Zahir. Lustre: A scalable, high performance file system. *Cluster File Systems, Inc*, 2002.

[18] Jiannong Cao, Yudong Sun, Xianbin Wang, and Sajal K Das. Scalable load balancing on distributed web servers using mobile agents. *Journal of Parallel and Distributed Computing*, 63(10):996–1005, 2003.

[19] Valeria Cardellini, Michele Colajanni, and Philip S Yu. Redirection algorithms for load sharing in distributed web-server systems. In *Distributed Computing Systems, 1999. Proceedings. 19th IEEE International Conference on*, pages 528–535. IEEE, 1999.

[20] Roger D. Chamberlain, Mark A. Franklin, et al. Gemini: An optical interconnection network for parallel processing. *IEEE Transactions on Parallel and Distributed Systems*, 13(10):1038–1055, 2002.

[21] Peter M Chen, Edward K Lee, Garth A Gibson, Randy H Katz, and David A Patterson. Raid: High-performance, reliable secondary storage. *ACM Computing Surveys (CSUR)*, 26(2):145–185, 1994.

[22] Kristina Chodorow. *MongoDB: the definitive guide*. " O'Reilly Media, Inc.", 2013.

[23] Hyunsik Choi, Jihoon Son, YongHyun Cho, Min Kyoung Sung, and Yon Dohn Chung. Spider: a system for scalable, parallel/distributed evaluation of large-scale rdf data. In *Proceedings of the 18th ACM conference on Information and knowledge management*, pages 2087–2088. ACM, 2009.

[24] Marek Chrobak and Lawrence L Larmore. An optimal on-line algorithm for k servers on trees. *SIAM Journal on Computing*, 20(1):144–148, 1991.

[25] Shyam C Deshmukh and Sudarshan S Deshmukh. Improved load balancing for distributed file system using self acting and adaptive loading data migration process. In *4th International Conference on Reliability, Infocom Technologies and Optimization (ICRITO)(Trends and Future Directions), 2015*, pages 1–6. IEEE, 2015.

[26] Bin Dong, Xiuqiao Li, Qimeng Wu, Limin Xiao, and Li Ruan. A dynamic and adaptive load balancing strategy for parallel file system with large-scale i/o servers. *Journal of Parallel and Distributed Computing*, 72(10):1254–1268, 2012.

[27] Jack Dongarra, Hans Meuer, and Erich Strohmaier. Top500 supercomputing sites. `http://www.top500.org`, 2016.

[28] Amos Fiat, Yuval Rabani, and Yiftach Ravid. Competitive k-server algorithms. In *Foundations of Computer Science, 1990. Proceedings., 31st Annual Symposium on*, pages 454–463. IEEE, 1990.

[29] Lester R Ford Jr and Delbert R Fulkerson. A simple algorithm for finding maximal network flows and an application to the hitchcock problem. Technical report, DTIC Document, 1955.

[30] Jim Gao. Machine learning applications for data center optimization. 2014.

[31] Alexandra Glagoleva and Archana Sathaye. Load balancing distributed file system servers: a rule-based approach. *Web-Enabled Systems Integration: Practices and Challenges: Practices and Challenges*, page 274, 2002.

[32] Gluster. Gluster-cloud storage for the modern data center, 2017.

[33] Richard Henwood. Components of a lustre filesystem - hpdd community space - hpdd community wiki. `https://wiki.hpdd.intel.com/display/PUB/Components+of+a+Lustre+filesystem`. (Accessed on 04/21/2017).

[34] John E Hopcroft and Richard M Karp. An nˆ5/2 algorithm for maximum matchings in bipartite graphs. *SIAM Journal on computing*, 2(4):225–231, 1973.

[35] Sikder Huq, Zubair Shafiq, Sukumar Ghosh, Amir Khakpour, and Harkeerat Bedi. Distributed load balancing in key-value networked caches.

[36] Intel. Lustre software release, 2017.

[37] K. R. Krish, Ali Anwar, and Ali R. Butt. hatS: A Heterogeneity-Aware Tiered Storage for Hadoop. In *The 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2014.

[38] K. R. Krish, Ali Anwar, and Ali R. Butt. Sched: A Heterogeneity-Aware Hadoop Workflow Scheduler. In *Proc. 22nd IEEE/ACM MASCOTS*, Paris, France, Sep. 2014.

[39] Linux. sar-linux man page, 2017.

[40] LLNL. Hacc i/o benchmark summary, 2017.

[41] LLNL. Ior benchmark summary, 2017.

[42] J. F. Martinez and E. Ipek. Dynamic multicore resource management: A machine learning approach. *IEEE Micro*, 29(5):8–17, 2009.

[43] Rich Miller. Google using machine learning to boost data center efficiency — data center knowledge, 2014.

[44] Esteban Molina-Estolano, Carlos Maltzahn, and Scott Brandt. Dynamic load balancing in ceph. 2008.

[45] AB MySQL. Mysql database server. *Internet WWW page, at URL: http://www. mysql. com (last accessed/1/00)*, 2004.

[46] James Oly and Daniel A Reed. Markov model prediction of i/o requests for scientific applications. In *Proceedings of the 16th international conference on Supercomputing*, pages 147–155. ACM, 2002.

[47] Victor Powell. Markov chains.

[48] Yingjin Qian, Eric Barton, Tom Wang, Nirant Puntambekar, and Andreas Dilger. A novel network request scheduler for a large scale storage system. *Computer Science - Research and Development*, 23(3):143–148, 2009.

[49] Cédric Richard, José Carlos M Bermudez, and Paul Honeine. Online prediction of time series data with kernels. *IEEE Transactions on Signal Processing*, 57(3):1058–1067, 2009.

[50] Ramesh R Sarukkai. Link prediction and path analysis using markov chains. *Computer Networks*, 33(1):377–386, 2000.

[51] Andrea Schaerf, Yoav Shoham, and Moshe Tennenholtz. Adaptive load balancing: A study in multi-agent learning. *Journal of Artificial Intelligence Research*, 2:475–500, 1995.

[52] R. H. Shumway and D. S. Stoffer. An approach to time series smoothing and forecasting using the em algorithm. *Journal of Time Series Analysis*, 3(4):253–264, 1982.

[53] Hyogi Sim, Youngjae Kim, Sudharshan S Vazhkudai, Devesh Tiwari, Ali Anwar, Ali R Butt, and Lavanya Ramakrishnan. Analyzethis: an analysis workflow-aware storage system. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 20. ACM, 2015.

[54] Aameek Singh, Madhukar Korupolu, and Dushmanta Mohapatra. Server-storage virtualization: integration and load balancing in data centers. In *Proceedings of ACM/IEEE SC*, 2008.

[55] Shane Snyder, Philip Carns, Kevin Harms, Robert Ross, Glenn K Lockwood, and Nicholas J Wright. Modular hpc i/o characterization with darshan. In *Proceedings of the 5th Workshop on Extreme-Scale Programming Tools*, pages 9–17. IEEE Press, 2016.

[56] Splunk. Operational intelligence, log management, application management, enterprise security and compliance — splunk. `https://www.splunk.com`. (Accessed on 04/21/2017).

[57] Kenton Varda. Protocol buffers: Googles data interchange format. *Google Open Source Blog, Available at least as early as Jul*, 2008.

[58] Feiyi Wang, Sarp Oral, Saurabh Gupta, Devesh Tiwari, and Sudharshan S Vazhkudai. Improving large-scale storage system performance via topology-aware and balanced data placement. In *Parallel and Distributed Systems (ICPADS), 2014 20th IEEE International Conference on*, pages 656–663. IEEE, 2014.

[59] Feng Wang, Qin Xin, Bo Hong, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Tyce T McLarty. File system workload analysis for large scale scientific computing applications. In *Proceedings of the 21st IEEE/12th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 139–152, 2004.

[60] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320. USENIX Association, 2006.

[61] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, and Carlos Maltzahn. Crush: Controlled, scalable, decentralized placement of replicated data. In *Proceedings of ACM/IEEE SC*, 2006.

[62] Bing Xie, Jeffrey Chase, David Dillow, Oleg Drokin, Scott Klasky, Sarp Oral, and Norbert Podhorszki. Characterizing output bottlenecks in a supercomputer. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 8:1–8:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[63] G Peter Zhang. Time series forecasting using a hybrid arima and neural network model. *Neurocomputing*, 50:159–175, 2003.

[64] Dongfang Zhao, Zhao Zhang, Xiaobing Zhou, Tonglin Li, Ke Wang, Dries Kimpe, Philip Carns, Robert Ross, and Ioan Raicu. Fusionfs: Toward supporting data-intensive scientific applications on extreme-scale high-performance computing systems. In *2014 IEEE International Conference on Big Data (Big Data)*, pages 61–70. IEEE, 2014.