Toward Scalable Monitoring on Large-Scale Storage for Software Defined Cyberinfrastructure

Arnab K. Paul
Virginia Tech

Ryan Chard
Argonne National Laboratory

Kyle Chard
University of Chicago

Steven Tuecke
University of Chicago

Ali R. Butt
Virginia Tech

Ian Foster
Argonne and University of Chicago

ABSTRACT

As research processes become yet more collaborative and increasingly data-oriented, new techniques are needed to efficiently manage and automate the crucial, yet tedious, aspects of the data lifecycle. Researchers now spend considerable time replicating, cataloging, sharing, analyzing, and purging large amounts of data, distributed over vast storage networks. Software Defined Cyberinfrastructure (SDCI) provides a solution to this problem by enhancing existing storage systems to enable the automated execution of actions based on the specification of high-level data management policies. Our SDCI implementation, called Ripple, relies on agents being deployed on storage resources to detect and act on data events. However, current monitoring technologies, such as *inotify*, are not generally available on large or parallel file systems, such as Lustre. We describe here an approach for scalable, lightweight, event detection on large (multi-petabyte) Lustre file systems. Together, Ripple and the Lustre monitor enable new types of lifecycle automation across both personal devices and leadership computing platforms.

ACM Reference format:
Arnab K. Paul, Ryan Chard, Kyle Chard, Steven Tuecke, Ali R. Butt, and Ian Foster. 2017. Toward Scalable Monitoring on Large-Scale Storage for Software Defined Cyberinfrastructure. In *Proceedings of PDSW-DISCS'17: Second Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems, Denver, CO, USA, November 12–17, 2017 (PDSW-DISCS'17),* 6 pages.
DOI: 10.1145/3149393.3149402

1 INTRODUCTION

The data-driven and distributed nature of modern research means scientists must manage complex data lifecycles, across large-scale and distributed storage networks. As data scales increase so too does the overhead of data management—a collection of tasks and processes that are often tedious and repetitive, such as replicating, cataloging, sharing, and purging data. Software Defined Cyberinfrastructure (SDCI) [5] can drastically lower the cost of performing many of these tasks by transforming humble storage devices into "active" environments in which such tasks are automatically executed in response to data events. SDCI enables high-level policies to be defined and applied to storage systems, thereby facilitating automation throughout the end-to-end data lifecycle. We have previously presented a prototype SDCI implementation, called Ripple [4], capable of performing various actions in response to file system events.

Ripple empowers scientists to express and automate mundane data management tasks. Using a simple If-Trigger-Then-Action rule notation, users program their storage devices to respond to specific events and invoke custom actions. For example, one can express that when files appear in a specific directory of their laboratory machine they are automatically analyzed and the results replicated to their personal device. Ripple supports inotify-enabled storage devices (such as personal laptops); however inotify is not often supported on large-scale or parallel file systems. To support large-scale file systems we have developed a scalable monitoring solution for the Lustre [13] file system. Our monitor exploits Lustre's internal metadata capabilities and uses a hierarchical approach to collect, aggregate, and broadcast data events for even the largest storage devices. Using this monitor Ripple agents can consume site-wide events in real time, enabling SDCI over leadership class computing platforms.

In this paper we present our scalable Lustre monitor. We analyze the performance of our monitor using two Lustre file systems: an Amazon Web Service deployment and a high performance deployment at Argonne National Laboratory's (ANL) Leadership Computing Facility (ALCF). We show that our monitor is a scalable, reliable, and light-weight solution for collecting and aggregating file system events such that SDCI can be applied to multi-petabyte storage devices.

The rest of this paper is organized as follows: Section 2 presents related work. Section 3 discusses the SDCI concept and our implementation, Ripple. Section 4 describes our scalable monitor. We evaluate our monitor in Section 5 before presenting concluding remarks and future research directions in Section 6.

2 RELATED WORK

SDCI and data-driven policy engines are essential for reliably performing data management tasks at scale. A common requirement for these tools is the reliable detection of trigger events. Prior efforts in this space have applied various techniques including implementing data management abstraction layers and reliance on applications to raise events. For example, the integrated Rule-Oriented Data System [11] works by ingesting data into a closed data grid such that it can manage the data and monitor events throughout the data

PDSW-DISCS'17, Denver, CO, USA
© 2017 ACM. 978-1-4503-5134-8/17/11…$15.00
DOI: 10.1145/3149393.3149402

lifecycle. Other SDCI-like implementations rely on applications to raise trigger events [1].

Monitoring distributed systems is crucial to their effective operation. Tools such as MonALISA [9] and Nagios [2] have been developed to provide insight into the health of resources and provide the necessary information to debug, optimize, and effectively operate large computing platforms. Although such tools generally expose file system status, utilization, and performance statistics, they do not capture and report individual file events. Thus, these tools cannot be used to enable fine-grained data-driven rule engines, such as Ripple.

Other data-driven policy engines, such as IOBox [12], also require individual data events. IOBox is an extract, transform, and load (ETL) system, designed to crawl and monitor local file systems to detect file events, apply pattern matching, and invoke actions. Like the initial implementation of Ripple, IOBox is restricted to using either inotify or a polling mechanism to detect trigger events. It therefore cannot be applied at scale to large or parallel file systems, such as Lustre.

Monitoring of large Lustre file systems requires explicitly designed tools [8]. One policy engine that leverages a custom Lustre monitor is the Robinhood Policy Engine [7]. Robinhood facilitates the bulk execution of data management actions over HPC file systems. Administrators can configure, for example, policies to migrate and purge stale data. Robinhood maintains a database of file system events, using it to provide various routines and utilities for Lustre, such as tools to efficiently find files and produce usage reports. Robinhood employs a centralized approach to collecting and aggregating data events from Lustre file systems, where metadata is sequentially extracted from each metadata server by a single client. Our approach employs a distributed method of collecting, processing, and aggregating these data. In addition, our monitor publishes events to any subscribed listener, allowing external services to utilize the data.

3 BACKGROUND: RIPPLE

SDCI relies on programmable agents being deployed across storage and compute devices. Together, these agents create a fabric of smart, programmable resources. These agents can be employed to monitor the underlying infrastructure, detecting and reporting data events of interest, while also facilitating the remote execution of actions on behalf of users. SDCI is underpinned by the same concepts as Software Defined Networking (SDN). A separation of data and control planes enables the definition of high-level, abstract rules that can then be distributed to, and enforced by, the storage and compute devices comprising the system.

Ripple [4] enables users to define custom data management policies which are then automatically enforced by participating resources. Management policies are expressed as If-Trigger-Then-Action style rules. Ripple's implementation is based on a deployable agent that captures events and a cloud service that manages the reliable evaluation of rules and execution of actions. An overview of Ripple's architecture is depicted in Figure 1.

Architecture: Ripple comprises a cloud-based service plus a light-weight agent that is deployed on target storage systems. The agent is responsible for detecting data events, filtering them against active rules, and reporting events to the cloud service. The agent also provides an execution component, capable of performing local actions on a user's behalf, for example running a container or performing a data transfer with Globus [3].

A scalable cloud service processes events and orchestrates the execution of actions. Ripple emphasizes reliability, employing multiple strategies to ensure events are not lost and that actions are successfully completed. For example, agents repeatedly try to report events to the service. Once an event is reported it is immediately placed in a reliable Simple Queue Service (SQS) queue. Serverless Amazon Lambda functions act on entries in this queue and remove them once successfully processed. A *cleanup* function periodically iterates through the queue and initiates additional processing for events that were unsuccessfully processed.

Rules: Ripple *rules* are distributed to agents to inform the event filtering process and ensure relevant events are reported. A Ripple rule consists of a *trigger* and an *action*. The trigger specifies the conditions under which the action will be invoked. For example, a user may set a rule to trigger when an image file is created in a specific directory of their laptop. An action specifies the type of execution to perform (such as initiating a transfer, sending an email, running a docker container, or executing a local bash command, to name a few), the agent on which to perform the action, and any necessary parameters. These simple rules can be used to implement complex pipelines whereby the output of one rule triggers a subsequent action.

Event Detection: Ripple uses the Python Watchdog module to detect events on the local file systems. Using tools such as inotify and kqueue, Watchdog enables Ripple to function over a wide range of operating systems. As rules are registered with an agent users also specify the path to be monitored. The agent employs "Watchers" on each directory relevant to a rule. As events occur in a monitored directory, the agent processes them against the active rules to determine whether the event is relevant and warrants reporting to the cloud service.

Limitations: A key limitation of Ripple is its inability to be applied, at scale, to large storage devices (i.e., those that are not supported by Watchdog). Further, our approach primarily relies on targeted monitoring techniques, such as inotify, where specific directories are monitored. Thus, Ripple cannot enforce rules which are applied to many directories, such as site-wide purging policies.

Relying on targeted monitors presents a number of limitations. For example, inotify has a large setup cost due to its need to crawl the file system to place *watchers* on each monitored directory. This is both time consuming and resource intensive, often consuming a significant amount of unswappable kernel memory. Each watcher requires 1Kb of memory on a 64-bit machine, meaning over 512MB of memory is required to concurrently monitor the default maximum (524,288) directories.

We have explored an alternative approach using a polling technique to detect file system changes. However, crawling and recording file system data is prohibitively expensive over large storage systems.

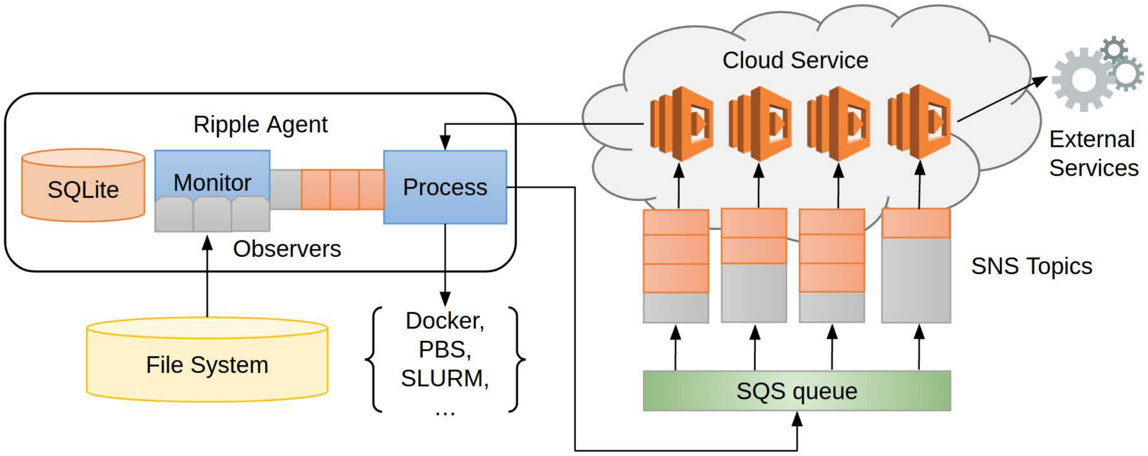

Figure 1: Ripple architecture. A local agent captures and filters file events before reporting them to the cloud service for processing. Actions are routed to agents for execution.

4 SCALABLE MONITORING

Ripple requires scalable monitoring techniques in order to be applied to leadership class storage systems. To address this need we have developed a light-weight, scalable monitor to detect and report data events for Lustre file systems. The monitor leverages Lustre's internal metadata catalog to detect events in a distributed manner and aggregates them for evaluation. The monitor produces a complete stream of all file system events to any subscribed device, such as a Ripple agent. The monitor also maintains a rotating catalog of events and an API to retrieve recent events in order to provide fault tolerance.

Like other parallel file systems, Lustre does not support inotify; however, it does maintain an internal metadata catalog, called "ChangeLog." An example ChangeLog is depicted in Table 1. Every entry in a ChangeLog consists of the record number, type of file event, timestamp, date, flags, target File Identifier (FID), parent FID, and the target name. Lustre's ChangeLog is distributed across a set of Metadata Servers (MDS). Actions which cause changes in the file system namespace or metadata are recorded in a single MDS ChangeLog. Thus, to capture all changes made on a file system our monitor must be applied to all MDS servers.

Our Lustre monitor, depicted in Figure 2, employs a hierarchical publisher-subscriber model to collect events from each MDS ChangeLog and report them for aggregation. This model has been proven to enable scalable data collection solutions, such as those that monitor performance statistics from distributed Lustre storage servers [10]. One *Collector* service is deployed for each MDS. The Collector is responsible for interacting with the local ChangeLog to extract new events before processing and reporting them. Events are reported to a single *Aggregator* for prosperity and publication to consumers.

File events, such as creation, deletion, renaming, and attribute changes, are recorded in the ChangeLog as a tuple containing a timestamp, event type, parent directory identifier, and file name. Our monitor collects, aggregates, and publishes these events using three key steps:

(1) **Detection:** Events are initially extracted from the ChangeLog by a Collector. The monitor will deploy multiple Collectors such that each MDS can be monitored concurrently. Each new event detected by a Collector is required to be processed prior to being reported.
(2) **Processing:** Lustre's ChangeLog uses parent and target file identifiers (FIDs) to uniquely represent files and directories. These FIDs are not useful to external services, such as Ripple agents, and must be resolved to absolute path names. Therefore, once a new event is retrieved by a Collector it uses the Lustre *fid2path* tool to resolve FIDs and establish absolute path names. The raw event tuples are then refactored to include the user-friendly paths in place of the FIDs before being reported.
(3) **Aggregation:** A publisher-subscriber message queue (ZeroMQ [6]) is used to pass messages between the Collectors and the Aggregator. Once an event is reported to the Aggregator it is immediately placed in a queue to be processed. The Aggregator is multi-threaded, enabling it to both publish events to subscribed consumers and store the events in a local database with minimal overhead. The Aggregator maintains this database and exposes an API to enable consumers to retrieve historic events.

Collector's are also responsible for purging their respective ChangeLogs. Each Collector maintains a pointer to the most recently extracted event and can therefore clear the ChangeLog of previously processed events. This ensures that events are not missed and also means the ChangeLog will not become overburdened with stale events.

5 EVALUATION

We have deployed our monitor over two Lustre testbeds to analyze the performance, overheads, and bottlenecks of our solution. Before investigating the monitor's performance we first characterize the capabilities of the testbeds to determine the rate at which events are generated. Using a specifically built event generation script, we

Table 1: A Sample ChangeLog Record.

Event ID	Type	Timestamp	Datestamp	Flags	Target FID	Parent FID	Target Name
13106	01CREAT	20:15:37.1138	2017.09.06	0x0	t=[0x200000402:0xa046:0x0]	p=[0x200000007:0x1:0x0]	data1.txt
13107	02MKDIR	20:15:37.5097	2017.09.06	0x0	t=[0x200000420:0x3:0x0]	p=[0x61b4:0xca2c7dde:0x0]	DataDir
13108	06UNLNK	20:15:37.8869	2017.09.06	0x1	t=[0x200000402:0xa048:0x0]	p=[0x200000007:0x1:0x0]	data1.txt

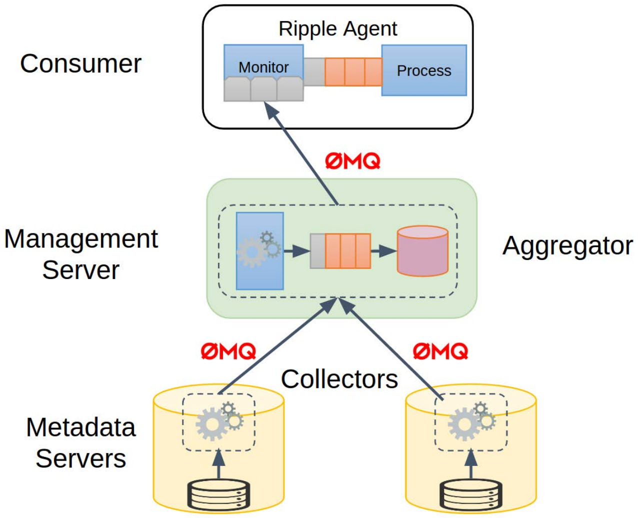

Figure 2: The scalable Lustre monitor used to collect, aggregate, and publish events to Ripple agents.

apply the monitor under high load to determine maximum throughput and identify bottlenecks. Finally, we use file system dumps from a production 7PB storage system to evaluate whether the monitor is capable of supporting very large-scale storage systems.

5.1 Testbeds

We employ two testbeds to evaluate the monitor's performance. The first testbed, referred to as AWS, is a cloud deployment of Lustre using five Amazon Web Service EC2 instances. The deployment uses Lustre Intel Cloud Edition, version 1.4, to construct a 20GB Lustre file system over five, low-performance, t2.micro instances and an unoptimized EBS volume. The configuration includes two compute nodes, a single Object Storage Service (OSS), an MGS, and one MDS.

The second testbed provides a larger, production-quality, storage system. This testbed, referred to as Iota, uses Argonne National Laboratory's Iota cluster's file system. Iota is one of two pre-exascale systems at Argonne and comprises 44 compute nodes, each with 72 cores and 128GB of memory. Iota's 897TB Lustre store leverages the same high performance hardware and configuration (including four MDS) as the 150PB store planned for deployment with the Aurora supercomputer. However, it is important to note that at present, the file system is not yet configured to load balance metadata across all four MDS, thus these tests were performed with just one MDS.

As a baseline analysis we first compare operation throughput on each file system. We use a Python script to record the time taken to create, modify, or delete 10,000 files on each file system. The performance of these two parallel file systems differs substantially, as is shown in Table 2. Due to the low-performance nature of the instances comprising the AWS testbed (t2.micro), just 352 files could be written per second. A total of 1366 events can be generated per second. As expected, the performance of the Iota testbed significantly exceeded this rate. It is able to create over 1300 files per second and more than 9500 total events per second.

Table 2: Testbed Performance Characteristics.

	AWS	Iota
Storage Size	20GB	897TB
Files Created (events/s)	352	1389
Files Modified (events/s)	534	2538
Files Deleted (events/s)	832	3442
Total Events (events/s)	1366	9593

5.2 Results

To investigate the performance of our monitor we use the Python script to generate file system events while our monitor extracts them from an MDS ChangeLog, processes them, and reports them to a listening Ripple agent. To minimize the overhead caused by passing messages over the network, we have conducted these tests on a single node. The node is also instrumented to collect memory and CPU counters during the tests to determine the resource utilization of the collection and aggregation processes.

Event Throughput: Our event generation script combines file creation, modification, and deletion to generate multiple events for each file. Using this technique we are able to generate over 1300 events per second on AWS and more than 9500 events per second on Iota.

When generating 1366 events per second the AWS-based monitor is capable of detecting, processing, and reporting just 1053 to the consuming Ripple agent. Analysis of the monitor's pipeline shows that the throughput is primarily limited by the preprocessing step following events being extracted from a ChangeLog. This is due to in part to the low-performance, t2.micro instance types used in the testbed. When experimenting on the Iota testbed we found the monitor is able to process and report, on average, 8162 events per second. This is 14.91% lower than the maximum event generation rate achieved on the testbed. Although this is an improvement over the AWS testbed, we found the overhead to be caused by the repetitive use of the *fid2path* tool when resolving an event's absolute path. To alleviate this problem we plan to process events in batches, rather than independently, and temporarily cache path mappings to minimize the number of invocations. Another limitation with this experimental configuration is the use of a single MDS. If the fid2path resolutions were distributed across multiple MDS, the throughput of the monitor would surpass the event generation rate. It is important to note that there is no loss of events once they have been processed, meaning the aggregation and reporting steps introduce no additional overhead.

Monitor Overhead: We have captured the CPU and memory utilization of the Collector, Aggregator, and a Ripple agent consumer processes. Table 3 shows the peak resource utilization during the Iota throughput experiments. These results show the CPU cost of operating the monitor is small. The memory footprint is due to the use of a local store that records a list of every event captured by the monitor. In a production setting we could further limit the size of this local store, which would in turn reduce the overall resource usage. We conclude that when using an appropriate maximum store size, deploying these components on the physical MDS and MGS servers would induce negligible overhead on their performance.

Table 3: Maximum Monitor Resource Utilization.

	CPU (%)	Memory (MB)
Collector	6.667	281.6
Aggregator	0.059	217.6
Consumer	0.02	12.8

5.3 Scaling Performance

Understanding the throughput of the monitor only provides value when put in the context of real-world requirements. Thus, we analyzed NERSC's production 7.1PB GPFS file system, called tlproject2. This system has 16,506 users and over 850 million files. We analyzed file system dumps from a 36 day period and compared consecutive days to establish the number of files that are created or changed each day. It is important to note that this method does not represent an accurate value for the number of times a file is modified, as only the most recent file modification is detectable, and also does not account for short lived files.

As shown in Figure 3, we found a peak of over 3.6 million differences between two consecutive days. When distributed over a 24 hour period this equates to just 42 events per second. Assuming a worst-case scenario where all of these events occur within an eight hour period results in approximately 127 events per second, still well within the monitor's performance range. Although only hypothetical, if we assume events scale linearly with storage size, we can extrapolate and expect Aurora's 150PB to generate 25 times as many events, or 3,178 events per second, which is also well within the capabilities of the monitor. It should be noted that this estimate could significantly underestimate the peak generation of file events. Further online monitoring of such devices is necessary to account for short lived files, file modifications, and the sporadic nature of data generation.

6 CONCLUSION

SDCI can resolve many of the challenges associated with routine data management processes enabling researchers to automate many of the tedious tasks they must perform. In prior work we presented a system for enabling such automation, however it was designed using libraries commonly available on personal computers but not often available on large-scale storage systems. Our scalable Lustre monitor addresses this shortcoming and enables Ripple to be used on some of the world's largest storage systems. Our results show that the Lustre monitor is able to detect, process, and report thousands of events per second—a rate sufficient to meet the predicted needs of the forthcoming 150PB Aurora file system.

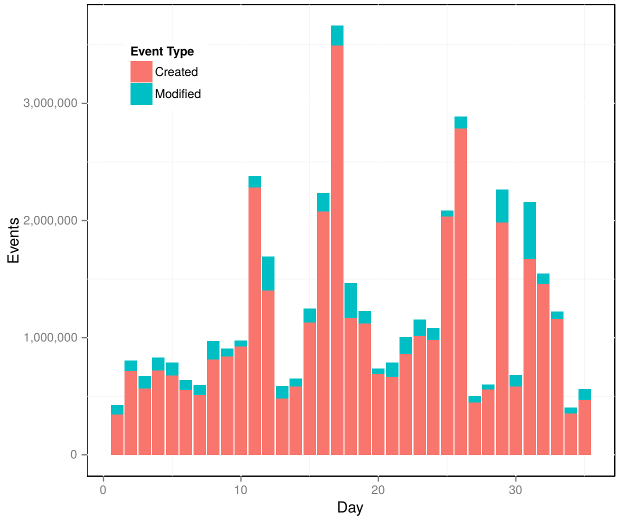

Figure 3: The number of files created and modified on NERSC's 7.1PB GPFS file system, tlproject2, over a 35 day period.

Our future research focuses on investigating monitor performance when using multiple distributed MDS, exploring and evaluating different message passing techniques between the collection and aggregation points, and comparing performance against Robinhood in production settings. We will also further investigate the behavior of large file systems to more accurately characterize the requirements of our monitor. Finally we are actively working to deploy Ripple on production systems and in real scientific data management scenarios, in so doing we are demonstrating the value of SDCI concepts in scientific computing platforms.

ACKNOWLEDGMENTS

This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357. We also acknowledge generous research credits provided by Amazon Web Services. This work is also sponsored in part by the NSF under the grants: CNS-1565314, CNS-1405697, and CNS-1615411.

REFERENCES

[1] M. AbdelBaky, J. Diaz-Montes, and M. Parashar. Software-defined environments for science and engineering. *The International Journal of High Performance Computing Applications*, page 1094342017710706, 2017.

[2] W. Barth. *Nagios: System and network monitoring.* No Starch Press, 2008.

[3] K. Chard, S. Tuecke, and I. Foster. Efficient and secure transfer, synchronization, and sharing of big data. *IEEE Cloud Computing*, 1(3):46–55, 2014.

[4] R. Chard, K. Chard, J. Alt, D. Y. Parkinson, S. Tuecke, and I. Foster. RIPPLE: Home Automation for Research Data Management. In *The 37th IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2017.

[5] I. Foster, B. Blaiszik, K. Chard, and R. Chard. Software Defined Cyberinfrastructure. In *The 37th IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2017.

[6] P. Hintjens. *ZeroMQ: messaging for many applications.* " O'Reilly Media, Inc.", 2013.

[7] T. Leibovici. Taking back control of HPC file systems with Robinhood Policy Engine. *arXiv preprint arXiv:1505.01448*, 2015.

[8] R. Miller, J. Hill, D. A. Dillow, R. Gunasekaran, G. M. Shipman, and D. Maxwell. Monitoring tools for large scale systems. In *Proceedings of Cray User Group Conference (CUG 2010)*, 2010.

[9] H. B. Newman, I. C. Legrand, P. Galvez, R. Voicu, and C. Cirstoiu. Monalisa: A distributed monitoring service architecture. *arXiv preprint cs/0306096*, 2003.

[10] A. K. Paul, A. Goyal, F. Wang, S. Oral, A. R. Butt, M. J. Brim, and S. B. Srinivasa. I/o load balancing for big data hpc applications. In *5th IEEE International Conference on Big Data(Big Data)*, 2017.

[11] A. Rajasekar, R. Moore, C.-y. Hou, C. A. Lee, R. Marciano, A. de Torcy, M. Wan, W. Schroeder, S.-Y. Chen, L. Gilbert, P. Tooby, and B. Zhu. iRODS Primer: Integrated rule-oriented data system. *Synthesis Lectures on Information Concepts, Retrieval, and Services*, 2(1):1–143, 2010.

[12] R. Schuler, C. Kesselman, and K. Czajkowski. Data centric discovery with a data-oriented architecture. In *1st Workshop on The Science of Cyberinfrastructure: Research, Experience, Applications and Models*, SCREAM '15, pages 37–44, New York, NY, USA, 2015. ACM.

[13] P. Schwan et al. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the 2003 Linux symposium*, volume 2003, pages 380–386, 2003.