Consistency and Fault Tolerance Considerations for the Next Iteration of the DOE Fast Forward Storage and IO Project

Jay Lofstead
Sandia National Laboratories
gflofst@sandia.gov

Ivo Jimenez
University of California, Santa Cruz
ivo@cs.ucsc.edu

Carlos Maltzahn
University of California, Santa Cruz
carlosm@soe.ucsc.edu

***Abstract*—The DOE Extreme-Scale Technology Acceleration Fast Forward Storage and IO Stack project is going to have significant impact on storage systems design within and beyond the HPC community. With phase 1 of the project complete, it is an excellent opportunity to evaluate many of the decisions made to feed into the phase 2 effort. With this paper we not only provide a timely summary of important aspects of the design specifications but also capture the underlying reasoning that is not available elsewhere.**

The initial effort to define a next generation storage system has made admirable contributions in architecture and design. Formalizing the general idea of data staging into burst buffers for the storage system will help manage the performance variability and offer additional data processing opportunities outside the main compute and storage system. Adding a transactional mechanism to manage faults and data visibility helps enable effective analytics without having to work around the IO stack semantics. While these and other contributions are valuable, similar efforts made elsewhere may offer attractive alternatives or differing semantics that could yield a more feature rich environment with little to no additional overhead. For example, the Doubly Distributed Transactions (D^2T) protocol offers an alternative approach for incorporating transactional semantics into the data path. Another project, PreDatA, examined how to get the best throughput for data operators and may offer additional insights into further refinements of the Burst Buffer concept.

This paper examines some of the choices made by the Fast Forward team and compares them with other options and offers observations and suggestions based on these other efforts. This will include some non-core contributions of other projects, such as some of the demonstration metadata and data storage components generated while implementing D^2T, to make suggestions that may help the next generation design for how the IO stack works as a whole.

I. Introduction

Current production HPC IO stack design is unlikely to offer sufficient features and performance to adequately serve the needs of an extreme scale platform. To address these limitations, a joint effort between the US Department of Energy's Office of Advanced Simulation and Computing and Advanced Scientific Computing Research commissioned an effort to develop a design and prototype for an IO stack suitable for the extreme scale environment. This is a joint effort led by Lawrence Livermore National Laboratory, with the DOE Data Management Nexus leads Rob Ross and Gary Grider as coordinators and contract lead Mark Gary. The participating labs are LLNL, SNL, LANL, ORNL, PNL, LBNL, and ANL. Additional industrial partners contracted include the Intel Lustre team, EMC, DDN, and the HDF Group. This team has developed a specification set [8] for a future IO stack to address the identified challenges. The first phase recently completed with a second phase currently getting underway as of early 2014. The core focus of the first phase was basic functionality and design. Overriding many of the decisions during this and any subsequent phases is the reality of budgets. Placing GBs of NVRAM on every node, while a potentially advantageous approach, is not financially feasible. With this in mind, the second phase will refine this design incorporating fault recovery and other features missing from the first phase.

The overall design seeks to offer high availability, byte-granular, multi-version concurrency control. Through the use of a copy-on-write style mechanism, multiple versions of an object can be stored in potentially greatly reduced space. It assumes the client interface will be through an IO library affording a more complicated interface that offers richer functionality requiring only minimal end-user code changes. Managing most data access in a platform-local layer rather than requiring writing to centralized storage will better support the performance and energy requirements of extreme scale integrated application workflows. At a more detailed view, the various layers of the IO stack each contribute different functionality and performance implications.

The overall target machine architecture is illustrated in Figure 1.

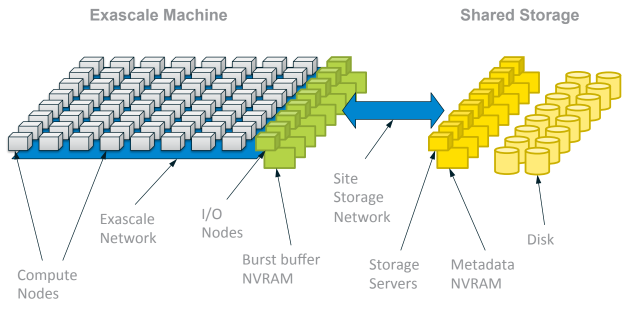

Fig. 1. Target Machine Architecture

The basic architecture incorporates five layers. The top layer is a high level IO library, such as the demonstration

HDF5 library [21]. The intent is to only have access to the storage stack through such an API to manage the complexity of working with the lower layers. For HDF5, the Virtual Object Layer (VOL) interface is used to replace the default functioning, such as working through MPI-IO, to instead use the new stack functionality. Below the user API is an IO forwarding layer that redirects IO calls from the compute nodes to the IO dispatching layer. This IO forwarding layer is analogous to the function of the IO nodes in a BlueGene machine. The next two layers have considerable functionality. The IO dispatcher (IOD) serves as the primary storage interface for the IO stack and is the only way to reach the persistent storage array in lower layers. The Distributed Application Object Storage (DAOS) layer serves as the persistent storage interface and is intended to be the foundation on which everything else is built with no dependence on any technologies specified above it. For example, the IOD layer and burst buffers, using additional storage outside of the storage array to stage and possibly process data prior to moving it to storage, are not required for DAOS to operate properly. At the bottom is the Versioning Object Storage Device (VOSD). It serves as the interface for storing objects of all types efficiently.

The focus of this paper is primarily the IOD layer given the critical role it has in the performance and functionality of the entire stack. Most of the key features explored in this paper all have a strong presence in the IOD layer motivating the focus of this examination.

The core idea for IOD is to provide a way to manage the IO load that is separate from the compute nodes and the storage array. Communication intensive activities, such as data rearrangement, can be moved to the IOD layer offloading the communication load from the compute nodes. IOD has three main purposes. First, if the optional burst buffer is available, it works as a fast cache absorbing write operations for the slower trickle out to the central storage array. It can also be used to retrieve objects from the central storage array for more efficient read operations and offers data filtering to make client reads more efficient. Second, it offers the transaction mechanism for controlling data set visibility and to manage faults that would prevent a data set from being used. Third, data processing operations can be placed in the IOD. These operations are intended to offer data rearrangement, filtering, and similar operations prior to data reaching the central storage array.

While these ideas are not necessarily new, they are new twists on best of class efforts for these technologies. For example, offloading the collective two-phase data sieving from the compute nodes to reorganize data has proven effective at reducing the total time for writing data due to fewer participants involved in the communication patterns [16]. Beyond these broad items, there are many important details some of which are examined in more detail below.

Along with the analysis of the published design documents, a discussion of the design philosophy representing the overall intent is presented. This information represents information that may or may not have been written down, but is the intent of ultimate product. These insights were gained based on personal conversations with the team members [4], [11], [3] discussing some of the potential challenges with the design as written. These ideas are presented to give a fuller picture of where the project is going rather than dwelling on the limitations of the published documents.

The rest of the paper is organized as follows. First, an overview of the target machine architecture and how it maps to the proposed components is presented in Section II. Section III provides a more detailed overview of the IOD layer and discusses some of the features of incorporating burst buffers as designed and suggests some considerations and alternatives for the next generation of this project. Section IV discusses the transactions approach offered in the IOD layer and the corresponding epochs in the DAOS layer. It also offers a comparison to the D^2T system given the very similar high-level design and motivating use case. Section V discusses the system overall with recommendations on what design elements should be considered based on broader issues with current HPC data centers. The paper is concluded in Section VI with a summary of the broad issues covered in the paper.

II. Overview

The demonstration system uses some particular existing technology to enable testing the proposed concepts. The mapping is presented in Figure 2. We examine this mapping by evaluating the descriptive blocks at the bottom of the figure.

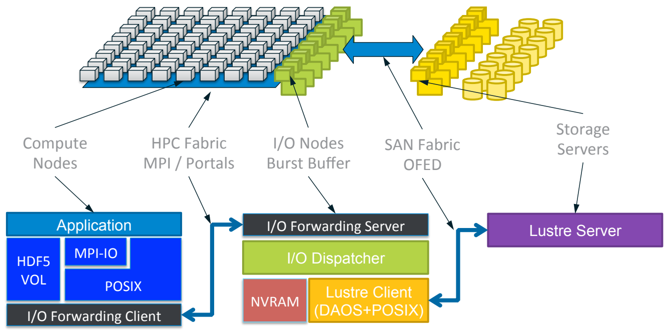

Fig. 2. Mapping FFSIO Components to Machine Architecture

The dark blue block that includes the HDF5 VOL block represents the User API layer. It is worth noting that while this is the intended interaction mechanism, it is still possible for other interfaces like MPI-IO and POSIX to interact with the system. The IO Forwarding layer is represented by the two black boxes. The represented split shows that the IO nodes can offer some control over how data is moved from the compute area towards storage. The IO dispatcher, the primary focus of the evaluation presented in this paper, is shown in the light green box. The one extension is the dark pink block representing technology like SSDs on the IO nodes. This is where a burst buffer, if the NVRAM is available, would be hosted. The DAOS layer is the yellow box. The VOSD is the purple box.

The specification of HDF5 and Lustre at the top and bottom of the stack is an expedience for demonstrating the functionality. In a production version of the proposed system, these may be represented by other technologies.

Throughout the paper, multiple terms are used to define an operation applied to data outside of the main application computation. These include both function shipping and just the term operator. In all cases, the idea is a piece of code is placed at some location along the IO path processing data as it passes through that point. Note that this works in both directions as well as in place. For example, when reading, an operator may be used to only retrieve the pieces of a large data set from the DAOS and VOSD layers into the IOD caches automatically. It is also possible that an operator is deployed to either rearrange data or generate a new copy with different ordering. For example, changing the fast dimension for a multi-dimensional array can greatly improve reading performance for particular reading patterns. In all cases, these operators offer a chance to apply a relatively light piece of processing to data rather than an intensive data analytics operation. These heavy-weight operations should be left to the main computation resources leaving capacity for other concurrent applications using the IOD layer.

III. IOD Layer and Burst Buffers

The IOD layer is intended to function as a higher performance, machine local caching layer for the DAOS/VOSD layers. Even in a simplified system, by keeping the IOD layer as part of the stack, making portable code is simpler since it eliminates the need to use a different lower level APIs from the user layer (e.g., HDF5) if the system lacks a feature, such as burst buffers. In an ideal design, the IOD API could function solely an directly on the DAOS servers. The reality is that performance demands and budget constraints will force something like burst buffers to compensate for the performance afforded and to offer a location to process data down to volumes that can fit through the limited bandwidth. Given the criticality of burst buffers and the transaction mechanism for end-to-end performance, these ideas are examined in detail below with comparison to alternatives that may have been avoided due to perceived performance limitations. In particular, the D^2T protocol was designed to address these sorts of scenarios and has demonstrated excellent performance.

The idea of burst buffers were initially explored in the context of data staging [2], [1], [17], [24]. These initial designs all use extra compute nodes to represent the data storage buffer given the lack of any dedicated hardware support for this functionality. The desired outcome of these initial studies is to motivate how such functionality might be incorporated and the potential benefits. Later, these concepts were proposed to be incorporated as part of the IO stack [18], [6], [5]. The current Fast Forward IOD design recommends incorporating SSDs, but specifically lists these devices as optional. Unfortunately, not incorporating burst buffers and the use of SSDs in the IOD layer may be problematic. First, the IOD design currently is written assuming burst buffers. This means that the bulk of IO operations will only hit the IOD layer and proposed functionality, such as the function shipping, do not discuss the impact on the DAOS layer or the operators themselves should a burst buffer not be available. Consider the important functionality of data rearrangement and doing things like changing the fast array dimension on shared, spinning media. The design of DAOS assumes that it will only be involved when persisting a completed transaction and only for a fraction of the total transactions created. Transaction frequency at the IOD layer can be higher since IOD does not "flatten" transactions. Flattening is the process of transferring the set of copy-on-write changes from transaction/epoch n to transaction $n + m$ into the DAOS layer. The DAOS layer mirrors the copy-on-write functionality, but stores the differences between epochs rather than individual transactions. The primary difference is that at the DAOS layer, data is stored in large, logically contiguous chunks to manage the metadata load and hopefully improve read performance.

One of the bigger concerns is that the original data staging proposals all used compute nodes while the newer proposals seek not only to make them a fixed portion of the IO stack, but also shared across all machine users. The PreDatA [24] paper in particular examines the potential costs and advantages of where to place operators similar to the IOD proposed function shipping. There are two key takeaways from PreDatA. First, placement matters. Depending on the communication intensity vs. computation intensity, where along the IO path to place the operation can matter significantly. Second, and more importantly, the amount of time spent processing for the operators was stretched to the point where it consumed nearly all of the time between IO operations. The given ratios of compute processes to staging process examined is representative for future extreme scale platforms. If anything, the ratios offer more staging processes than IOD processes would be available.

In the case of the written IOD design, it describes a fixed-sized staging area that is partitioned on a per-application basis. This is unlikely to be useful because of the limited compute and communication capacity to spare to perform these operations at a bottleneck in the IO path. The use of a separate data staging area intentionally separate from the IO path allows using operators on limited resources leaving the IO path clear for strictly data movement. A nuance of this design is discussed in the Design Philosophy below.

By concentrating the Burst Buffers and function shipping into the storage stack, three problems arise. First, the amount of network bandwidth, IO bandwidth, and compute power consumed for example operations from a single application is likely to completely monopolize the IOD processes. Second, if space and time partitioning is used instead, the functionality risks being too small to be useful. For example, if some of the IO nodes are dedicated either throughout an application run (space partitioning) or for just phases of the application runtime (time partitioning), the allocated capacity or ownership duration may be insufficient to be useful. Third, the long-term hardware performance advantage for SSDs is questionable. Recent studies have shown that the erase-before-write and interference between reading and writing with flash-based SSDs can cause severe performance problems [20]. The inclusion of an optional use SSD layer in the new Trinity machine at Los Alamos will offer a test bed to determine how likely these observed problems would affect a production extreme scale platform.

Current NAND-based flash devices top out at around 400 MB/sec. The key spec that is missing from this number is that 400 MB/sec is a measure of the fixed number of available IOPS multiplied by the block size. This represents the ideal streaming performance possible. The problem is that it costs an IOP to read 1 byte or 1 block (4KB or 8KB, depending on the device). It costs 1 IOP to write a full block–usually. In some cases, it will cost 2 IOPs. This accounts for the required pre-erase write prior to writing to a reused block. In the worst case, it can be 3 or more IOPS per write. For example, if writing forces both garbage collection and block compaction, many blocks may have to be read, the data combined, the blocks erased, the combined data written, and then the intended write operation can finally proceed on the newly freed blocks (erase and then write). One-third of 400 MB/sec, about 133 MB/sec, is well below the streaming performance of HDDs. Granted, there is still rotational and seek latency to deal with for HDDs, but the advantage for SSDs has evaporated and potentially turned into a considerable penalty at a cost premium. There are faster SSD solutions on the market that incorporate DRAM for caching and using the PCIe bus, for example, but their price precludes them from use in an extreme scale platform given budget constraints.

Given these features, the optionality and even incorporation of burst buffers in the current design should be carefully considered. Much of the advanced, key functionality proposed as they are currently designed ultimately relies on the existence of burst buffers to work. Further thought about how to have an IOD layer both with and without a burst buffer is required before they can be considered optional. As the design stands today, they are a required part of the IOD layer for proper functioning. Unfortunately, it is not clear that they can address the performance concerns they are intended to cover.

A. *Design Philosophy*

The burst buffers design, as presented in the IOD documents, limits the placement of the function operators and SSD buffers to the IO nodes. The team does acknowledge the limitations of this design and intend to ultimately focus on spreading the IOD layer from the IO nodes into the compute area as well. This is intended both to help address the limitations of the IO bandwidth and compute capability of these few nodes for data processing, but also to take advantage of new layers in the storage hierarchy. By incorporating NVRAM into compute nodes, new options for buffering data prior to being moved to centralized storage become available and addresses some of the concerns about SSD performance. For example, including a small amount of Phase Change memory into many or most compute nodes offers a way to move data outside of both the compute and IO path for data and communication intensive operations. Other projects [24] have suggested this will have value, but the cost will have to be considered as part of the overall platform budget. This lessens the impact of some operators while offering additional options for places to store data.

Burst buffers being optional is a high level goal, but not one considered at a detailed level within the design. For example, if there is no burst buffer, all of the advanced functionality proposed for the IOD layer would have to work against the DAOS layer instead. For example, function shipping assumes it will operate on fast, local data within the IOD layer rather than against the globally shared DAOS layer. With the additional desire to support using compute node resources for these operations, serious work will be required to make a fully functional end-to-end IOD layer implementation for a production system.

Another concern that is acknowledged, but no thought has been applied to, is the requirement that a single IOD process of the set assigned to an application is the master for any operation. Should the number of concurrent applications exceed the available nodes, sharing an IOD process will be required. The requirements both in terms of scheduling and resource management were cut from the project due to funding limitations. Since partitioning of the IOD processes for exclusive use by particular applications is the assumed operating mode, should insufficient IOD resources be available, either a job could be delayed or IOD resources could be reallocated from a different process could be redeployed for use by the new job. Handling resilience concerns for the IOD processes must also be address. These sorts of considerations still need to be made for a full production system.

IV. Transactions, Epochs, and Metadata

The transaction mechanism manifests in two forms. At the IOD layer, they are called transactions and are used to judge whether or not a set of distributed, asynchronous modifications across a set of related objects is complete or not. It is also used to control access by treating the transaction ID of committed transaction as a version identifier. At the DAOS layer, they are called epochs and represent persisted (durable) transactions from the IOD layer. Each of these offers different functionality, but are connected as is explained below. How these differ from the D^{2}T approach is also explored. While IOD's and D^{2}T's transactions are seemingly very different, they use a similar high-level design, but very different implementation, to solve the same problem.

A. *IOD Transactions*

To understand how transactions are used in the IOD layer, some terminology and concepts must be explained first. At the coarsest grain level is a container. Each container provides the single access context through which to access a collection of objects. Transactions are used to treat a series of modifications to the objects within a single container atomically. Each transaction is simply an arbitrary sequence of read and write operations by a single application to that single container that is ultimately treated as an atomic action. Each transaction must be managed through the end-user API with begin and end calls to bracket the set of operations. Conceptually, containers corresponds to a something akin to an HDF5 file in a traditional file system. The objects in each container represent different data within a file. The three initially defined object types are key-value stores, multi-dimensional arrays, and blobs. The easiest way to understand these types is to evaluate these from the perspective of an HDF5 file, the initial user interface layer. The key-value store represents a collection of attributes or groups. The array represents a potentially multi-dimensional array. The blob represents a byte stream of arbitrary contents. The fundamental difference between an array and a blob is that the array has metadata specifying the dimension(s). At the physical layer within the IO nodes, all of these objects may be striped across multiple IO nodes. Given this context, the transactions come in two forms.

First is a single leader transaction managed by IOD based on calls from a single client. The underlying assumption is that the client side will manage the transactional operations itself and the single client is capable of reporting to the IOD how to evolve the transaction state. Passive failures are detected based on the single client only. Any actively detected failure will have to be managed by this single client.

The second form is called multi-leader and has the IOD layer manage the transactions from a collection of clients. In this case, when the transaction is created, a count of clients is provided to the IOD layer. As clients commit their changes to the container, the reference count is reduced. Once the count reaches 0, the transaction is automatically committed. Failure detection is fully passive based on all participating clients.

In both cases, the end user must interact with the transaction mechanism through the end-user API (e.g., HDF5) to specify the transaction type and potentially specify a transaction ID.

1) Design Philosophy: Undocumented, but inherent in the design of these transactions is how faults are detected. The initial design assumes the current Lustre fault detection mechanism that can determine if a process or node is no longer reachable. This detection happens at the DAOS layer and when a fault is detected, the rollback process is pushed up to the IOD layer for all non-persisted or non-committed transactions. This defines how a fault will be detected and what will trigger a passive fault recovery (i.e., transaction abort).

There are two steps for beginning a transaction on a container. The first step is for one or more process to open the container. This handle can be shared eliminating the need for every participating process to hit the IOD layer to open the file. The second step is a call to determine how many leaders will participate in the transaction. In the single leader case, there is no aggregation of success/fail statuses to determine the final transaction state. Instead, it is assumed that the client will fully manage the transaction. In the multi-leader model, some subset from 2 to n where n is the count of all processes, declare themselves a leader for this container operation to the IOD layer. Any number of processes can participate in modifying container without regard to whether or not they are a leader. Once each leader has finished, with the assumption that any clients they may be responsible for are finished as well, the IOD layer aggregates those responses to either commit or abort the transaction.

Ultimately, with the passive detection of faults for transaction leaders, the transaction mechanism can work very well. A mostly unstated restriction that is being relaxed is that every sequential transaction on a container is considered dependent on the earlier transaction. Should one output be delayed and the subsequent five succeed, when the delayed process finally fails, all six transactions are rolled back. The thought of using this mechanism to store subsequent checkpoint outputs in the same container to both save space, but not care if one fails, cannot work in the current form. This has been acknowledged and is being relaxed requiring a new parameter to the creation of a transaction determining if it will be dependent or not.

B. DAOS Epochs

The Epoch mechanism differs from transactions. Instead of focusing on when a particular output is complete, an epoch represents incremental persisted copies of a container. To simplify the mapping between an IOD transaction and the DAOS epochs, when an IOD transaction is persisted to DAOS, the IOD transaction ID is the used as the epoch ID. The key difference is that at the DAOS layer, some transaction (epoch) IDs will not be represented since not all IOD transactions are necessarily persisted.

C. Metadata Management

Metadata management has been a perennial challenge for parallel storage systems. Eliminating metadata management as a special case and instead treating it just as data is a central design goal of the Fast Forward project. At one extreme, the “file” system provides a way to store objects returning IDs back to the user. The user is then responsible for understanding what the byte stream in that object means and how to identify it. Think of it like a numbered bank account that contains a certain amount of money, but the only identifier is a number rather than the name of an account holder. This introduces the burden of remembering what an object really is to the user layer and any communication of the object identity up to the user. There is abstract third-party entity, such as the naming serivce part of traditional file systems for a different user to search for an object. The main dangers are losing an ID means losing how to interpret the bytes in an object and no way for two applications to use the storage system to share data without communicating newly created IDs. At the other extreme are traditional file systems that offer object storage along with naming, authentication, and authorization services. In the former case, there is no overhead in the file system for maintaining any metadata associated with the stored byte stream. In the latter case, the file system offers a consistent way to handle it and includes tracking and identification services at a performance cost to maintain the metadata consistently. FFSIO uses a hybrid approach for metadata management that is half-way between providing no inherent metadata support and having a fully integrated, but separate metadata management system.

Eliminating metadata as a core component of a file system is not new. It has been explored as part of the Light Weight File Systems project [19]. In LWFS, the metadata service is explicitly limited to a user task with the storage layer limited to data storage/retrieval, authorization, and authentication. This approach proved feasible. Using this hybrid approach is less common [22] and introduces other issues.

IOD and DAOS both share a philosophy that they will have to maintain the metadata about how the physical pieces of the logical objects are striped and where they are placed. The primary metadata management is done at the DAOS layer with the IOD layer relying on the DAOS layer for all authoritative information about containers and objects. The only place where the IOD layer manages metadata for itself is to manage how the different objects are striped across the IO nodes.

1) Design Philosophy: While the metadata design is not fully defined, there are a few things that are intended. For example, there is a standard, well-known container that is the system metadata. This includes the list of all other containers. This container is treated like any other data in the system and striped as appropriate. Unfortunately, this still couples the metadata to a single object that must serialize access. If the metadata, including information about striping and other data layout operations were separated completely from the data path, more scalable throughput could be achieved. The real challenge of this is introduced by the IOD, DAOS, and VOSD layers collectively. Each of these requires some different metadata storage and the migration is transparent to the user. Supporting fully independent metadata with this model is difficult. Serious thought on how to do this effectively outside the data path should be considered for phase two.

Based on the lessons from the D^{2}T metadata service [13] construction and the prior experiments with LWFS, having a completely separate metadata service is feasible. Rather than making it a bottleneck in the IO path, it is another service that users must interact with if they need those services. Users can manage everything by maintaining the metadata including the list of objects themselves. However, there are drawbacks to this 100% client-side approach.

With a client-side only approach, there is a serious risk of the metadata service and the object store getting out of sync. While having a metadata-less object storage service is desirable, the different semantics from traditional file systems requires some considerations. In this case, should these services get out of sync, three particular risks are introduced. First, a client could create a dangling entry in the metadata service that does not correspond to any objects in the object store. Second, a client could create orphaned objects that have no associated metadata entries. Third, updates to the metadata or object store service should be an atomic operation, but due to a lack of coordination, a window where the system is inconsistent appears.

Ultimately, the consistency semantics required must be determined. If a metadata service is required and it must be in sync with the object storage service, then additional work must be performed. In traditional file systems, the metadata and object storage updates are atomic. With decoupling metadata from object storage, should this atomicity still be desired, it requires both the ability for the services to participate in a task that is part of a larger atomic operation and a higher-level mechanism to manage the atomic operation.

Overall, while additional work is required to maintain a client-side only metadata service, it eliminates any potential bottlenecks related to updating metadata related to the object storage. The burden of tracking striping and other metadata that has traditionally been part of the metadata associated with the file system will have to be maintained by the object storage service. The lack of a centralized, serialized bottleneck to store that information improves concurrency.

D. Comparison to Other Protocols

Alternatives, such as Paxos [12] algorithms like ZooKeeper [10], suffer from two limitations making them unsuitable for this environment. First, the distributed servers in Paxos systems are all distributed copies of each other that eventually become consistent. Given the scale we wish to address, a single node's memory is unlikely to be able to hold all of the data necessary for many operations at scale. They also do not have a guarantee for when consensus will be achieved without using slower synchronous calls. For the tight timing we wish to support, we need guarantees of when a consistent state has been achieved. Second, these systems also all assume that updates are initiated from a single client process rather than a parallel set of processes as is the standard in HPC environments.

Another effort to offer consistency and data integrity for the ZFS file system [23] covers some of the same territory. Instead of a focus on the processes all having a notion of completion as a transaction, this work focuses on the integrity of the data movement operations. We view this work as something that should be considered hand-in-hand with a transaction approach to ensure the integrity of the movement of the data in addition to the agreement of processes about the successful completion of a parallel operation.

The transactions and epochs approach offered by FFSIO is most similar to D^{2}T , but is a specialized implementation of a similar protocol. In this case, D^{2}T is general and must use more general techniques. The FFSIO transactions and epochs take advantage of simplifications based on the architecture and the very specific use.

D^{2}T uses a second layer of coordination on the client side that greatly increases the scalability by consolidating messages from clients into unique sets prior to sending to the overall coordinator. A gossip protocol [9] may appear sufficient for this purpose, but the delay of eventual consistency is strictly avoided with this protocol to ensure guarantees at particular states in the code. For example, if a failure occurs, the global understanding of the role of all processes is required in order for effective communication to occur for operations like creating sub-transactions or voting. In this case, the protocol can offer stronger statements about consistency than these protocols offer. These features offer a way to easily scale the transaction protocol given the guarantees we wish to offer. The IOD approach does nothing to address these concerns.

The details of D^{2}T is discussed next to offer a performance evaluation that may be slower than FFSIO transactions and epochs for synchronous due to the explicit messaging rather than relying on the hardware mechanisms. At scale, the FFSIO performance should be similar to the D^{2}T approach.

E. Comparison to D^{2}T

The D^{2}T project [15] sought to develop an efficient approach for handling ACID-style transactions in an environment with parallel clients and multiple servers (doubly distributed). Rather than being aimed solely at data movement operations, D^{2}T seeks to address the general problem of managing any operation with multiple clients and servers. Consider the management of the analysis/visualization area, potentially similar to the IOD concept. The transaction protocol is used to help manage resizing of the resource allocation to the various analysis and visualization components. For the purposes of this discussion, D^{2}T could also be used to manage changing how IOD processes and/or nodes are used without exposing these changes to the client processes prematurely. This has been described and analyzed previously [7].

The example metadata and data storage services created as part of the D^{2}T project have no dependencies between transactions that prevent visibility should an older version be incomplete. This additional, intentional requirement by IOD offers different functionality than D^{2}T's example services. In the case of D^{2}T, the functionality is less, but also avoids some of the concerns outlined below.

The second iteration of the protocol [14] fixed scalability issues and demonstrated a scalable client-side coordination model with excellent performance. The performance measured for a complex transaction with D^{2}T is illustrated in Figure 3. This performance is explored in detail in a previous paper [14]. Briefly, all of the client processes are split into groups. Each group is managed by a sub-coordinator. The sub-coordinators are all managed by an overall coordinator. Clients are expected to be able to determine the success of failure of any action they take. Rather than gathering directly from all clients to the coordinator, messages are aggregated through the tree with duplicate information removed reducing both the size and count of messages sent to the coordinator. Broadcasting messages to clients proceed down the tree through the sub-coordinators as well. The breakdown of the number of participants in each role is shown in Table I. For comparison, consider the Number of Sub-Coordinators equivalent to IOD processes. The Processes Per Sub-Coordinator represents the number of clients that use a particular IOD process. For these tests, we maintained a balanced distribution and always used at least two sub-coordinators to slow down the processing.

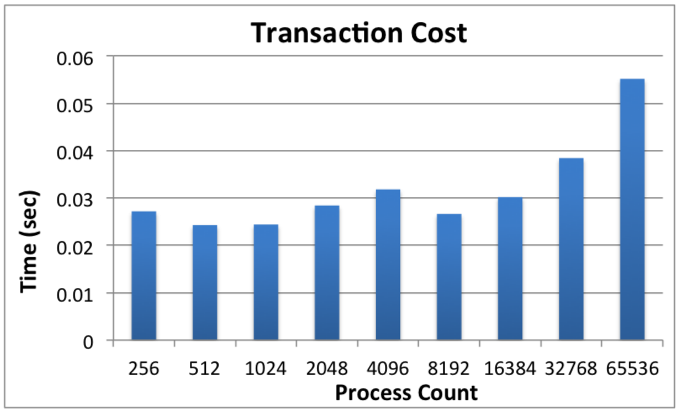

Fig. 3. Total Transaction Overhead

TABLE I. PERFORMANCE TESTS SCALING CONFIGURATION

Processes	Number of Sub-Coordinators	Processes Per Sub-Coordinator
256	2	128
512	2	256
1024	4	256
2048	8	256
4096	16	256
8192	32	256
16384	64	256
32768	128	256
65536	256	256

At a high level, both D^{2}T and the IOD transactions have the same design. In both cases, a hierarchical model is employed. In the case of D^{2}T, it is a purely client-side tree using semi-synchronous messaging. The messaging itself, in the current implementation, uses asynchronous MPI messages. The synchronous component comes from the timeout mechanism used to detect faults. It forces a level of coordination and synchronization for the protocol. For IOD, it is a server-side tree and fully asynchronous relying on the existing Lustre fault detection mechanism for failure detection. In both cases, there is a master in charge of managing the transaction and a collection of workers that aggregate into the master through second-level leaders. Beyond that, there are some significant differences. Some of the different choices made by IOD raise some possible concern.

The multi-leader model introduces the possibility of forcing a rollback of an entire transaction when a partial retry might be sufficient for success. Since the transactions are managed at a high level rather than the individual tasks, a failure in a limited distributed task can cause the entire transaction to fail. For example, consider 10 processes each have 5 tasks, but 3 of those 10 have an additional shared task to complete. If the task shared by the 3 processes fails on any of the three, the entire transaction would roll back because it is a coarse-grained success/failure. If a concept like sub-transactions at a task granularity were used, then it would be possible for the one process that failed to report just that failure. Then the transaction manager could reassign the resources for these 3 processes and try just that operation again. If it now succeeds, then the overall transaction can be marked successful only redoing the minimum amount of work required.

D^{2}T has addressed theses issues in a couple of ways. First, the sub-coordinators each have a list of processes from which they expect messages. Should a message be missed, it is noticed and corrective action can be taken. Second, D^{2}T has the concept of sub-transactions. The messaging requirements are illustrated in Figure 4. Sub-transactions represent finer grained operations than the entire output, D^{2}T can manage multiple writes per client by using a sub-transaction to represent the output for any item to the file (container). Because of how the sub-transactions are managed, the singleton sub-transactions, ones in which only a single process participates, must be declared before the transaction begins so that its existence can be broadcast as part of the begin transaction message. This ensures there is global knowledge that the sub-transaction is expected. That way if the coordinator (transaction leader) fails, whichever process takes over that role knows to expect a completion message for that sub-transaction or the overall transaction cannot complete. Global sub-transactions can be defined at any time since they are a global, synchronized operation broadcasting their existence. While this additional layer does introduce messaging, the overhead is quite small.

The advantages of eliminating these messages is not performance as demonstrated by the performance of D^{2}T. Instead, it offers a much less synchronous model that matches with different programming models, such as Charm++ or other task-based approaches. Since it can work for the bulk-synchronous model also, it is a more broadly applicable approach. This assumes that the observed potential issues can be addressed successfully.

V. BROADER DESIGN

At a broader level, there are some concerns that were partially clarified through conversations with the team. Consider a shared file system across an HPC data center. The current design maintains the metadata in its own container. Since copying data from the IOD layer to the DAOS layer requires an explicit persist call, how and when synchronizing

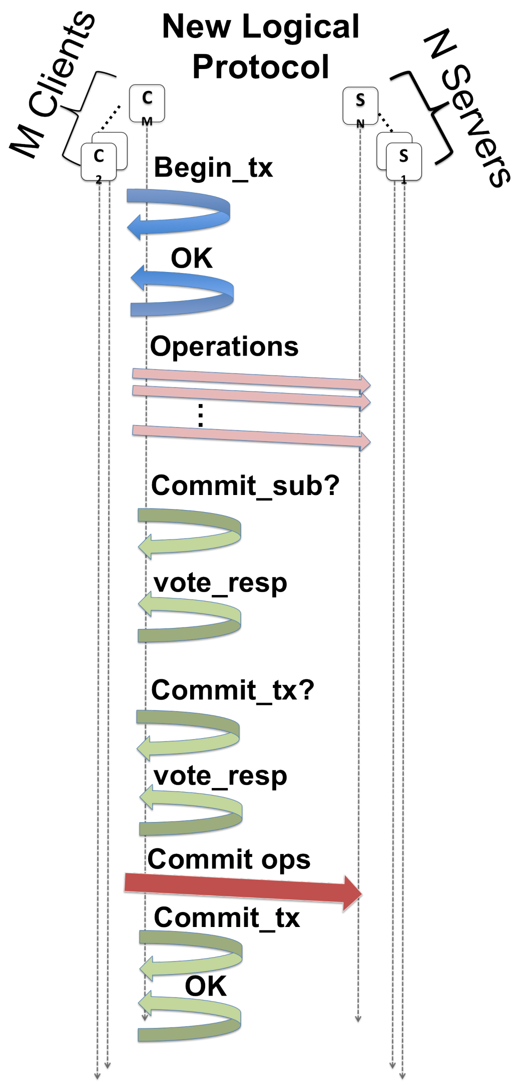

Fig. 4. Optimized Protocol

the metadata across the layers and potentially across machines occurs is undefined. Delaying synchronization until an explicit persist is called will reduce the update frequency, but delays the data visibility on other platforms. Ideally, the metadata object would need to be automatically persisted every time a container transaction is persisted to the DAOS layer.

The implication of this is that every transaction persist is double operation to account for the metadata persist. More importantly, the IOD-layer version of the metadata container may contain readable transactions that have not been persisted to the DAOS layer. How to handle this inconsistency between the two layers still needs to be explored.

A point of confusion rather than a potential design challenge is the change in definitions between the IOD layer and the DAOS layer. For the IOD layer, a container is a collection of objects. For the DAOS layer, a container is a collection of objects across a set of shards. For the IOD an object may be a shard of a global array. For DAOS, a shard can host a set of DAOS objects. Having the same names with locally correct, globally conflicting definitions serves to confuse how the system should work.

VI. Conclusions

The Fast Forward Storage and IO Stack project has designed a good first pass at addressing the requirements for an extreme scale data storage mechanism. The split between the IOD layer and the DAOS layer offers a fast place for intermediate data without requiring the overhead of writing to persistent storage. The envisioned transaction mechanism, while not perfect in the current form, is another good attempt to address both failures and prevent access to incomplete or incorrect data by downstream data consumers. Integrated with the IOD functionality, this concept represents the consensus approach for what will be required.

The partial metadata management incorporated into the IOD layer and the lack of consideration for how to handle and recover from failures are oversights in the current documents. It is our understanding that these will be addressed in the next phase and we hope to help inform that effort with our experiences.

We hope that the efforts made in the D^2T, Lightweight File Systems, and other efforts to explore the requirements for this space, along with the analysis presented in this paper will prove useful for the next phase of the Fast Forward project.

VII. Acknowledgements

The authors would like to thank Eric Barton, John Bent, Gary Grider, and Quincey Koziol for their early review comments and discussions that clarified the details of the design, the intent of the design, and the future plans.

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

We wish to thank Eric Barton for the some of the figures used to illustrate the system architecture.

References

[1] H. Abbasi, J. Lofstead, F. Zheng, S. Klasky, K. Schwan, and M. Wolf. Extending i/o through high performance data services. In *Cluster Computing*, Luoisiana, LA, September 2009. IEEE International.

[2] H. Abbasi, M. Wolf, and K. Schwan. LIVE data workspace: A flexible, dynamic and extensible platform for petascale applications. In *CLUSTER '07: Proceedings of the 2007 IEEE International Conference on Cluster Computing*, pages 341–348, Washington, DC, USA, 2007. IEEE Computer Society.

[3] E. Barton. 2014-03-21. personal communication.

[4] J. Bent. 2014-03-21. personal communication.

[5] J. Bent, S. Faibish, J. Ahrens, G. Grider, J. Patchett, P. Tzelnic, and J. Woodring. Jitter-free co-processing on a prototype exascale storage stack. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pages 1–5, April 2012.

[6] J. Bent, G. Grider, B. Kettering, A. Manzanares, M. McClelland, A. Torres, and A. Torrez. Storage challenges at los alamos national lab. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pages 1–5, April 2012.

[7] J. Dayal, J. Cao, G. Eisenhauer, K. Schwan, M. Wolf, F. Zheng, H. Abbasi, S. Klasky, N. Podhorszki, and J. Lofstead. I/o containers: Managing the data analytics and visualization pipelines of high end codes. In *In Proceedings of International Workshop on High Performance Data Intensive Computing (HPDIC 2013) held in conjunction with IPDPS 2013*, Boston, MA, 2013. Best Paper Award.

[8] Fastforward storage and i/o stack design documents. Intel FastForward Wiki, February 2014. https://wiki.hpdd.intel.com/display/PUB/Fast+Forward+Storage+and +IO+Program+Documents.

[9] A. Ganesh, A.-M. Kermarrec, and L. Massoulie. Peer-to-peer membership management for gossip-based protocols. *Computers, IEEE Transactions on*, 52(2):139–149, 2003.

[10] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *In USENIX Annual Technical Conference*, 2010.

[11] Q. Koziol. 2014-03-21. personal communication.

[12] L. Lamport and K. Marzullo. The part-time parliament. *ACM Transactions on Computer Systems*, 16:133–169, 1998.

[13] J. Lofstead and J. Dayal. Transactional parallel metadata services for application workdflows. In *In Proceedings of High Performance Computing Meets Databases at Supercomputing*, 2012.

[14] J. Lofstead, J. Dayal, I. Jimenez, and C. Maltzahn. Efficient transactions for parallel data movement. In *The Petascale Data Storage Workshop at Supercomputing*, Denver, CO, November 2013.

[15] J. Lofstead, J. Dayal, K. Schwan, and R. Oldfield. D2t: Doubly distributed transactions for high performance and distributed computing. In *IEEE Cluster Conference*, Beijing, China, September 2012.

[16] J. Lofstead, R. Oldfiend, T. Kordenbrock, and C. Reiss. Extending scalability of collective io through nessie and staging. In *The Petascale Data Storage Workshop at Supercomputing*, Seattle, WA, November 2011.

[17] A. Nisar, W.-k. Liao, and A. Choudhary. Scaling parallel I/O performance through I/O delegate and caching system. In *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[18] P. Nowoczynski, N. Stone, J. Yanovich, and J. Sommerfield. Zest checkpoint storage system for large supercomputers. In *Petascale Data Storage Workshop, 2008. PDSW '08. 3rd*, pages 1 –5, nov. 2008.

[19] R. A. Oldfield, A. B. Maccabe, S. Arunagiri, T. Kordenbrock, R. Riesen, L. Ward, and P. Widener. Lightweight I/O for scientific applications. In *Proceedings of the IEEE International Conference on Cluster Computing*, Barcelona, Spain, Sept. 2006.

[20] D. Skourtis, D. Achlioptas, C. Maltzahn, and S. Brandt. High performance & low latency in solid-state drives through redundancy. In *Proceedings of the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, INFLOW '13, page 6:16:9, New York, NY, USA, 2013. ACM.

[21] The HDF Group. Hierarchical data format version 5, 2000-2014. http://www.hdfgroup.org/HDF5.

[22] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *OSDI'06*, Seattle, WA, Nov. 2006.

[23] Y. Zhang, A. Rajimwale, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. End-to-end data integrity for file systems: A zfs case study. In R. C. Burns and K. Keeton, editors, *FAST*, pages 29–42. USENIX, 2010.

[24] F. Zheng, H. Abbasi, C. Docan, J. Lofstead, S. Klasky, Q. Liu, M. Parashar, N. Podhorszki, K. Schwan, and M. Wolf. PreDatA - preparatory data analytics on Peta-Scale machines. In *In Proceedings of 24th IEEE International Parallel and Distributed Processing Symposium, April, Atlanta, Georgia*, 2010.