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

With phase 1 of the Fast Forward Storage and IO Stack project complete, it is an excellent opportunity to evaluate many of the decisions made to feed into the phase 2 effort. The initial effort to define a next generation file system has made admirable contributions in architecture and design. Formalizing the general idea of data staging as burst buffers for the file system will help manage the performance variability and offer the data processing opportunities outside the main compute and file 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 their contributions are valuable, similar efforts made elsewhere may offer attractive alternatives or differing semantics that would 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. The PreDatA project examined how to get the best throughput for data operators can offer some insights for refining the Burst Buffer concept.

This paper examines some of the choices made by the FastForward 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.

Categories and Subject Descriptors

D.4 [**Software**]: Operating Systems; D.4.7 [**Operating Systems**]: Organization and Design—*hierarchical design*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
PDSW13 November 18, 2013, Denver, CO, USA
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2505-9/13/11...$15.00.
http://dx.doi.org/10.1145/2538542.2538567

General Terms

Design, Performance

1. 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, the US Department of Energy 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 the Intel Lustre team, Los Alamos National Laboratory, EMC, DDN, and the HDF Group. This team has developed a specification set [6] for a future IO stack to address the identified challenges. The first phase recently completed with a second phase getting underway. The core focus of the first phase was basic functionality and design. The second phase will refine this design incorporating fault recovery and other features missing from the first phase.

The overall design seeks to offer a byte-granular, multiversion concurrency control. Through the use of a copy-on-write style mechanism, multiple versions of a file can be stored in potentially greatly reduced space. By assuming the client interface will be through an IO library affords a more complicated interface that offers richer functionality without requiring user code changes. By managing data in the IOD layer rather than requring writing to centralized storage will better support the performnce requirements of integrated application workflows. At a more detailed view, the various layers of the IO stack each contribute different functionality and performance implications.

The basic architecture incorporates five layers. The top layer is a high level IO library, such as the demonstration HDF-5 library. 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. 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 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 two-phase, data sieving IO's data rearrangement process, 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 offer 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 it reaching the central storage array.

While these ideas are not necessarily new, they are new twists on best of class efforts for these techologies. 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 [13]. Beyond these broad items, there are many important details some of which are examined in more detail below.

The bottom two layers are the Data Access Object Storage (DAOS) and Versioning Object Storage Device (VOSD). DAOS is the typical interface that represents the individual objects stored including the semantics related to "files". VSOD is the actual storage mechanism for the objects defined at the DAOS layer.

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.

Along with the analysis of the published design documents, a discussion of the design philosophy representing the overall intent is presented. This informtion reprents information that may or moy not have been written down, but is the intent of ultimate product. These insights were gained based on personal converations with the team members 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. Section 2 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 3 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 4 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 5 with a summary of the broad issues covered in the paper.

2. BURST BUFFERS

The idea of burst buffers were initially explored in the context of data staging [2, 1, 14, 16]. 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 [4, 3]. The current FastForward IOD design recommends incorporating SSDs, but specifically lists these devices as optional. Unfortunately, both incorporating burst buffers and the use of SSDs in the IOD layer may be problematic. First, with these burst buffers being optional, the semantics must be clarified of how IOD works must change to use DAOS to store the data directly rather than storing them locally until explicitly persisted by the user. The design of DAOS does not incorporate a high transaction rate mechanism. Conversely, it assumes that it will only be involved when persisting a completed transaction and only for a fraction of the total transactions created. Further, with function shipping offering the ability to change how the data is stored and arranged prior to it being written to DAOS, this functionality may also be optional depending on the existence of the burst buffer or slow the IO path to accommodate the data processing. Consider the important functionality of data rearrangement and doing things like changing the fast array dimension.

One of the bigger concerns is the observation 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 [16] 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 that work. 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 forces a staging area deployment and then shares the staging area across all users simultaneously. 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.

By concentrating this functionality 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. Third, the hardware performance advantage for SSDs is questionable. 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 IOPS per write. This would be for a partial block write (read the old block, erase, write the modified block). 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 penalty at a considerable 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 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.

2.1 Design Philosophy

The burst buffers design, as presented in the IOD documents, limits the placement of the function operators and SSDs for buffers to the IO nodes. The team does acknowledge the limitations of this design and intend to ultimately focus on spreading the IOD 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 capaibility 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. 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 with the design. While this forces some serious thought on how to adapt the functionality proposed for IOD to work directly on DAOS will have to be considered. 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 as been applied to is the requirement that a single IOD process be the master for any operation. Should the number of concurrent applications exceed the available nodes, sharing an IOD process will be required. Since partitioning of the IOD processes for exclusive use by particular applications is the proposed 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. These sorts of consideraitons still need to be made for a full production system.

3. TRANSACTIONS AND EPOCHS

The transaction mechanism manifests in two forms. At the IOD layer, they are called transactions and are used to judge whether or not an output is complete or not and control access through treating the transaction ID 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^2T approach is also explored. While IOD's and D^2T's transactions are seemingly very different, they use a similar high-level design, but very different implementation, to solve the same problem.

3.1 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 serves to host a transaction and contains a collection of objects. Conceptually, containers corresponds to a 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 store, array shard, and blob. The easiest way to understand these types is to evaluate these from the perspective of an HDF-5 file, the initial user interface layer. The key-value store represents a collection of attributes. The array shard represents part of a potentially multi-dimensional array. It is referred to as a shard because it is likely a small piece of a globally defined array. The blob represents a byte stream of arbitrary contents. The fundamental difference between an array shard and a blob is that the array shard has metadata identifying its portion within the global, logical space while the blob is simply a 1-dimensional array of bytes that is not shared across IOD nodes. Should an operation be deployed into IOD to manipulate an array within a container, it would operate on the array shards rather than on blobs. Given this context, the transactions come in two forms.

First is a single leader transaction where the IOD manages 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.

The second form is called multi-leader and has the IOD layer manage the transactions. In this case, when the transaction is created, a count of clients is provided to the IOD layer. As clients write to the container, the reference count is reduced. Once the count reaches 0, the transaction is automatically committed.

3.1.1 *Design Philosophy*

Undocumented, but inherent in the design of these transactions is how faults are detected. The initial design assumes that the current Lustre fault detection mechanism that can determine if a process or node is no longer reachable is used. This defines how a fault will be detected and what will trigger a passive fault recovery (i.e., transaction abort).

A new term, "Primary", is used to define the process(es) that contact the IOD layer directly to open a container. Unfortunately, this term is never used nor defined within any of the published documents. There are some very positive and negative performance implications of this approach. On the positive side, like PVFS, it is possible for a single process to open a container and share that handle with other processes reducing the load on the IOD layer. The key idea is that the single leader vs. multi-leader transaction mode is automatically defined based on the number of Primary processes that open a container. This greatly simplifies any additional client-side load in using the different styles of transactions. Also on the positive side, it offers the ability to have a subset of processes that are considered transaction participants affording reduced load on the IOD processes and more localized transaction management on the client side. On the negative side, for a fully IOD managed transaction, ALL processes must open the container forcing a potentially immense number of simultaneous calls to the IOD layer to open a container. This sort of onslaught is what is dealt with on metadata servers today with mass file opens. The reality is that this ends up having the same sort of performance implications as distributed denial of service attacks on the IOD processes.

Ultimately, with the passive detection of faults for Primaries, the transaction mechanism can work very well. A mostly unstated restriction that will be relaxed in the next phase 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 planned on being relaxed in phase two.

3.2 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.

3.3 Metadata Management

Eliminating metaata management as a special case and instead treating it just as data is a central design goal of the Fast Forward project. Unfortunately, it is not completely achieved.

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 [?]. 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 workable. Using this hybrid approach is less common and introduces other issues.

IOD and DAOS both share a philosophy that they will have to maintain the metadata about how objects are striped and where they are placed. It was decided that the additional data to track the namespace as well was a small enough additional increment that it was worth maintaining within the system.

3.3.1 *Design Philosophy*

While the metadata design is not fully defined, there are a few things that are intended as part of this. 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 at both the IOD and the DAOS layers. 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 the IOD, DAOS, and VOSD layers. 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, if not impossible. Serious thought on how to do this outside the data path effectively should be considered for phase two.

The D^2T project created a simplistic model for a metadata service that has different features. It is described below and has been published previously [10]. Some of the major differences include the following. First, D^2T's metadata service has a way to get a list of the objects in the metadata service. Since this was intended as a test system to examine D^2T performance, we did not put the effort into a making it a robust system. While it does not address having a file equivalence like the container concept, it does introduce a way to figure out what is available. Stored along with this object description is a list of the global dimensions for array objects. One observation made as part of creating this service is that this is insufficient for many engineering codes that use non-regular meshes. Second, for each object, a list of the equivalent to shards are kept with a link back to the master object and the offset and size for each dimension for that shard. This approach is not scalable because of the explosion in the process count. Using a striping approach like IOD does is superior. Third, there is no mapping to another layer with different object layout and counts.

Based on the lessons from the D^2T metadata service construction and the prior experiments with LWFS, having a completely separate metadata service is workable. Rather than making it a bottleneck in the IO path, it is another service that users must interact with if they need those services. Otherwise, it is 100% out of the way. 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 introduced. First, a client could create an entry in the metadata service that does not correspond to any objects in the object store. Second, a client could create 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, eliminating any potential bottlenecks related to updating metadata related to the object storage are eliminated. 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.

3.4 Comparison to D^{2}T

The D^{2}T project [12] 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 [5].

The example metadata and data storage services created as part of the D^{2}T project lack one of the key transaction management features used by IOD. The major difference is that there are 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 more minimal, but also avoids some of the concerns outlined below.

The second iteration of the protocol [11] 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 1. This performance is explored in detail in a previous paper [11]. The breakdown of the number of participants in each role is shown in Table 1. For comparison, consider the Number of Sub-Coordinators equivalent to an IOD process. The Processes Per Sub-Coordinator represents the number of clients that use a particular IOD. For these tests, we maintained a balanced distribution and always used at least two sub-coordinators to slow down the processing.

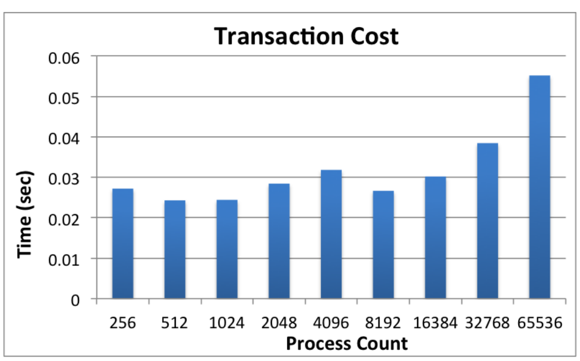

Figure 1: Total Transaction Overhead

At a high level, both D^{2}T and the IOD transactions have the same high-level 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

Table 1: 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

of coordination and synchronization for the protocol. For IOD, it is a server-side tree and fully asynchronous. 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.

First, should a transaction on a container not complete, all subsequent transactions on that container, even if they are complete and correct, will not be accessible. Once the transaction fails, all of these subsequent transactions will be aborted. While this is being reconsidered for phase two, it does complicate the interface to offer the additional transaction mode.

Second, in the multi-leader model, using a count of client connections to determine if transaction is complete is problematic. First, should this be in an environment where lost or repeated network messages are possible, then the count could be wrong inappropriately. This is a general problem and unlikely in the current environment, but is a concern on less integrated platforms. The idea of a count precludes any tracking of expected messages from sources introducing a potential hole. Second, the aggregation of completion messages are based on the local aggregation point reporting to the transaction leader the state. There is no ability to detect or recover from a failure of this IOD process that would report to the transaction leader. Third, because a simple count of participating processes is sent with the beginning of the transaction, each process is strictly limited to a single output operation. This precludes a process writing multiple array shards, for example. To put this in concrete terms, it means that a file could only contain a single, globally distributed variable (at the root level of HDF5) OR an attribute per process. If this count actually represents a begin/end transaction pair instead, it suffers from a lack of accountability for missing object (shard) writes. All it would indicate is that a process started and ended a transaction connection. There is no validation that a process has done everything required of it.

D^{2}T has addressed this count issue 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 2. 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 man-

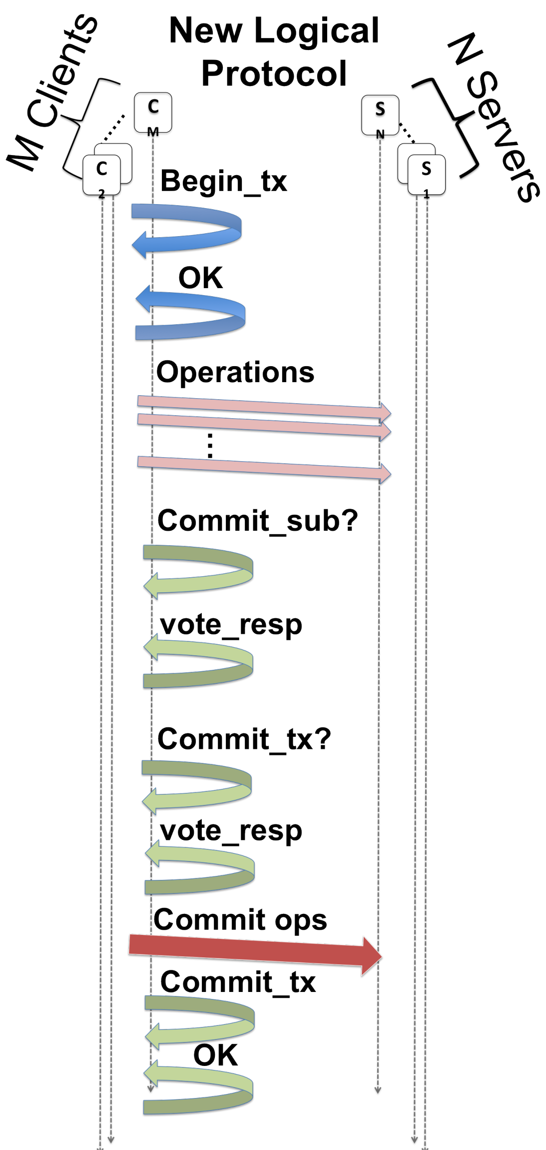

Figure 2: Optimized Protocol

aged, 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, which ever 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.

3.5 Comparison to Other Protocols

Alternatives, such as Paxos [9] algorithms like ZooKeeper [8], 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.

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 [7] 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.

Another effort to offer consistency and data integrity for the ZFS file system [15] 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.

4. 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 the metadata across the layers and potentially across machines occurs is undefined. By delaying the synchronization until an explicit persist is called will reduce the update frequency, it does delay the visibility of data. 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 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.

5. 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^{2}T, 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.

6. ACKNOWLEDGEMENTS

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.

7. 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] 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.

[4] 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.

[5] 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.

[6] Fastforward storage and i/o stack design documents. Intel FastForward Wiki, February 2014.

[7] 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.

[8] 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.

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

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.