Efficient Transactions for Parallel Data Movement

Jay Lofstead
Sandia National Laboratories
gflofst@sandia.gov

Jai Dayal
Georgia Institute of Technology
jdayal3@cc.gatech.edu

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

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

ABSTRACT

The rise of Integrated Application Workflows (IAWs) for processing data prior to storage on persistent media prompts the need to incorporate features that reproduce many of the semantics of persistent storage devices. One such feature is the ability to manage data sets as chunks with natural barriers between different data sets. Towards that end, we need a mechanism to ensure that data moved to an intermediate storage area is both complete and correct before allowing access by other processing components. The Doubly Distributed Transactions (D^2T) protocol offers such a mechanism. The initial development [9] suffered from scalability limitations and undue requirements on server processes. The current version has addressed these limitations and has demonstrated scalability with low overhead.

Categories and Subject Descriptors

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

General Terms

Design, Performance

1. INTRODUCTION

Data movement to a new location, including both persistent storage such as disk or to another compute area staging area, requires some mechanism to help determine when a data movement operation is both complete and correct. While the latter is a more complex problem that requires validation along the path both to and from any storage location, marking a data set as an atomic whole is more straightforward. Existing file systems use the concept of a file with locks against reading and writing, depending on the user and the operation, to achieve this semantic. This works well to offer a way for a user or programmer to identify a data set. The user simply looks for either a file or some additional metadata stored within a file to identify the relevant portions. For a data staging area in an HPC environment, pieces of the data set are being sent from a potentially very large collection of processes to a collection of data staging areas, similar to a parallel storage array. Given the ephemeral nature of data in a staging area, the overhead of parallel file system is onerous. Instead, simpler key-value style object stores with appropriate metadata for object location make more sense. The difficulty with this alternative structure is blocking a data set preventing premature or inappropriate access.

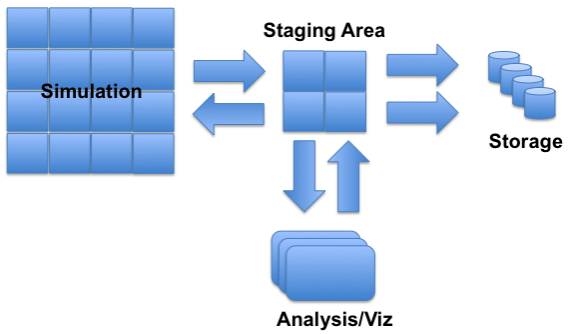

Figure 1: Staging Model

The kind of Integrated Application Workflow (IAW) environment we are targeting is illustrated in Figure 1. The idea for transactions is to handle the movement of data from the simulation to the staging area, the data processing in the analysis/visualization routine, and the subsequent movement of data to storage. In addition, the management of the analysis/visualization area is also considered. The transaction protocol is used to help manage resizing of the resource allocation to the various analysis and visualization components. This has been described and analyzed previously [4]. This additional use for Doubly Distributed Transactions (D^2T) demonstrates it is general enough to be used for operations beyond strictly data movement [4]. For this evaluation, we focus on the data movement case because it is the most intensive use case and demonstrates the possibilities at scale.

Alternatives, such as Paxos [7] algorithms like ZooKeeper [6], suffer from two limitations making them unsuitable for this

PDSW '13 November 17-21, 2013, Denver, CO, USA

ACM 978-1-4503-2505-9/13/11...$15.00.
http://dx.doi.org/10.1145/2538542.2538567

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 guaranteed 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. The Intel/Lustre FastForward epoch approach [2, 10] is discussed in Section 4.

The initial implementation of D^2T showed that it is possible to create a somewhat scalable MxN two-phase commit protocol, but it suffered from limitations. First, the server side was required to manage the transactions directly forcing inclusion of both a communication mechanism between the clients and the servers for control messages and the server processes had to incorporate coordination mechanisms to determine the transaction state among the server processes. This limited transactions to generally a single server application. Second, the single control channel between the clients and the servers did not serve to address scalability. Either the aggregate size and/or quantity of messages from clients to the client-side coordinator or the size of the messages between the client-side coordinator and the server-side coordinator will eventually overwhelm memory. To address these limitations, we redesigned the protocol. The design changes and new performance results are discussed in the next sections.

The rest of this paper is organized as follows. First, a discussion of the new protocol design is presented in Section 2. Next, some additional components required to demonstrate this system are discussed in Section 3. The evaluation is presented next in Section 4 followed by Section 5 with a discussion of alternatives. Finally, conclusions are presented in Section 6.

2. NEW PROTOCOL DESIGN

To summarize the semantics of D^2T for those unfamiliar with it, consider the following. Since we have parallel clients and servers (MxN), it is necessary to have a mechanism to manage each single operation by a collection of clients with a collection of servers to ensure that it is completed correctly. We must also have a way to manage the set of these operations as a single, atomic action. The difficulty with this scenario is the need to maintain global knowledge across all of the clients and servers. This global knowledge allows decisions across all clients. In a traditional distributed transaction, the single client always knows the state reported by all of the "server" resources used. When the client is split into multiple pieces, each piece ultimately needs to have the same knowledge, including any knowledge that might be local only. If this does not happen, no guarantees can be made. The message count explosion implied by this level of knowledge is daunting and what prompted this project. Adding complexity for data movement operations is that a typical data set will contain multiple globally distributed variables that each must be treated as an atomic unit.

To accommodate these requirements, we have the concept of a transaction representing the total set of operations and the sub-transaction representing a single operation. Each sub-transaction can represent either a single process performing an action (singleton) or all of the parallel clients participating in a single action (global). In either case, each sub-transaction is managed as part of the overall transaction. Voting and commit operations are performed both at the sub-transaction level and at the transaction level. With the re-implementation of the protocol, these semantics have not changed.

The performance of the old protocol was not acceptable. An example performance evaluation is presented in Figure 2. In this example, the 256 - 2 represents the number of clients and the number of servers. The best performance, at 256 clients, is a total of 1.2 seconds. The new protocol implementation has improved to about 0.07 seconds at worst (0.055 seconds on average), but at 65536 processes. While the new tests do not include data movement, the minor contribution of data output to the total time in these tests demonstrates that the problem was not data movement, but overhead in the protocol itself.

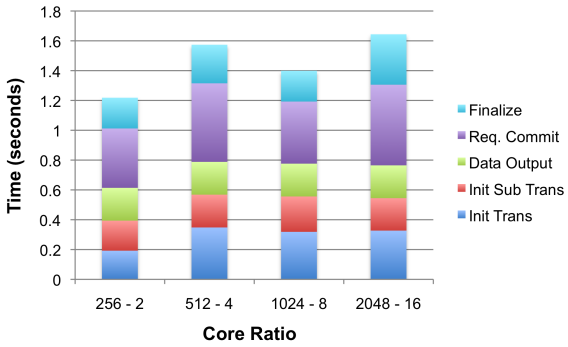

Figure 2: Full Protocol Performance

The redesign and re-implementation of the protocol focused on the lessons learned during the creation of the original full protocol. At a detailed level, four major changes were made.

First, an assumption is made that a client is capable of determining whether or not a server has completed an operation successfully or not. This is a typical feature for RPC mechanisms and is not seen as excessive, particularly when compared with the old requirement of integrating the protocol processing and the client-server communication channel. This eliminates much of the client-server communication requirements.

Second, the servers are only required to offer a mechanism to mark an operation as "in process" and prevent access for inappropriate clients. This may be handled by a transactional wrapper around a service instead.

Third, the server must offer a way to mark an operation as "committed" during the "commit" phase of the two-phase commit. Again, this may be implemented in a transactional wrapper. As an alternative, a scalable locking service like Chubby [3] might be employed.

Fourth, a second layer of coordination on the client side is introduced that greatly increases the scalability by consolidating messages from clients into unique sets prior to sending to the overall coordinator. A gossip protocol [5] 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

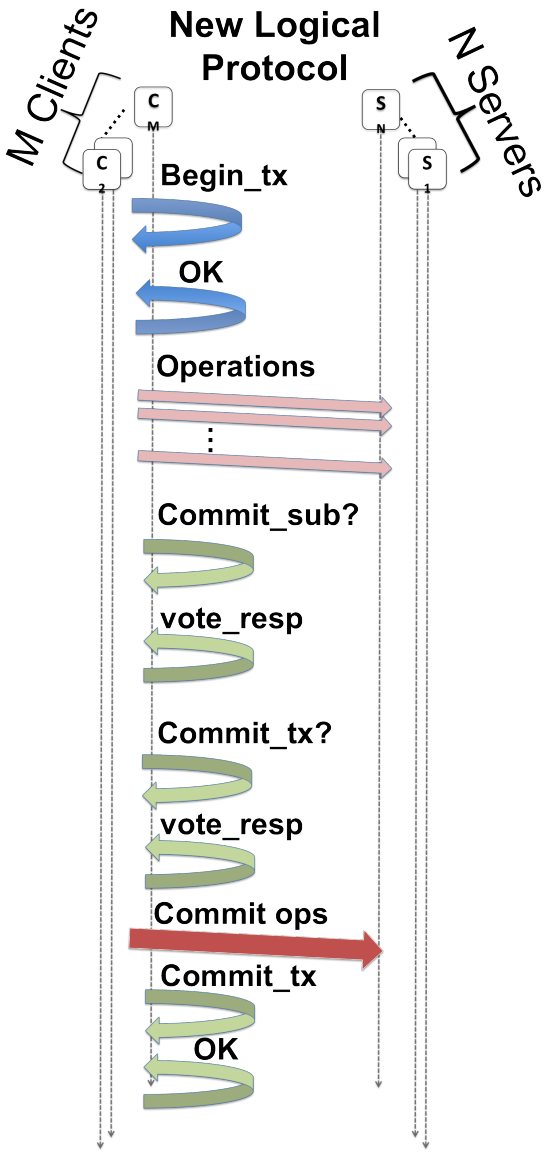

Figure 3: Optimized Protocol

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 key features of these changes comes down to client-side coordination only and layered messaging. The client-side only messaging eliminates the greatest performance penalty of polling for messages both from the client side as well on the server side to coordinate operations there. It also eliminates the need to try to re-negotiate a client-server connection in the case of a failure of the coordinator on either side.

The layered messaging with message consolidation offers a way to reduce both the message count volume and the aggregate size of the total number of messages. For example, when beginning a transaction, all singleton sub-transactions must be collected upwards, reconciled into a single global list of unique items, and then sent back out to all processes. This gather, reduce to unique items, broadcast structure is required to support failure recovery. This operation is also used for creating global sub-transactions and for final voting of the overall transaction. While other organizations for providing global consistency are available, the performance achieved with this approach is sufficiently good that we have not felt the need to investigate other approaches.

The failure recovery mechanism was initially discussed elsewhere [9]. It has since been redesigned to work within the framework of the hierarchical structure in the current implementation. In summary, timeouts for messages between participants is used to detect failures. While this functionality is both complete and tested, it is beyond the scope of this paper to describe in detail the requirements to make the recovery process work and to offer the performance impact of detecting and recovering from a failure. The performance achieved is a small fraction of a second longer than the timeout time. This work is being submitted elsewhere in full detail with a full evaluation.

3. REQUIRED ELEMENTS FOR IAWS

To better represent an IAW, a simple data staging area that offers transaction support with data storage and metadata operations is required. To offer this functionality, two services are created.

3.1 Datastore Service

The initial and key functionality for a data staging area is the ability to store data in a context separate from the original creating application and offer retrieval by others. While it may be physically stored on one of the same nodes as the application in some future cases particularly, this configuration is considered separate for the purposes of this discussion. The datastore service offers the simple ability to store and retrieve an object in the context of a transaction ID. The service takes a data blob, generates a unique ID, marks the data as part of a particular transaction and "in process" to hide it from other transactions. Service users can request objects by ID in the context of a transaction ID. The transaction ID filters the object information returned to only include objects that are either "active" or are part of the current transaction. In this way, the service can create an object hidden from view of other clients and still support the transaction activities. Once the transaction is voted correct and complete, the list of objects associated with a transaction ID can be marked as "active" making them visible to other clients. While other functionality, such as authorization controls, are necessary for a production system, this demonstration offers no additional functionality.

3.2 Metadata Service

The metadata requirements for the datastore service to be generally useful has been presented previously [8]. The details of what features are provided are described below. Since that publication, the implementation has been completed and used in the demonstration system. Many details have been determined fleshing out the requirements in more detail. Based on the examples explored, this service assumes a rectangular data domain meaning that the simulation space is some rectangular solid (i.e., a rectangular multi-dimensional array). Other configurations, such as an unstructured grid, are not supported. This and other configurations would have to be supported for any general system. Using the rectangular simulation domain, the metadata ser-

vice offers the ability to create a variable with some global dimensions and a collection of "chunks" representing the data from a single process or similar magnitude of data. As with the datastore service, the metadata service supports creating a variable in the context of a particular transaction ID and marking it as "in process" to prevent inappropriate access. This variable ID can then be distributed to the other participating processes for their creation of chunk entries. Alternatively, some aggregation scheme could be employed to reduce the load on the metadata service. Also like the datastore service, there is an activate function to mark a variable as "active" making it visible to other clients. Other concerns, such as authorization that are solely part of the metadata service rather than part of the transaction protocol itself, have similarly been omitted for the purposes of this demonstration.

3.3 Discussion

These services are separated for two primary reasons. First, the scalability of the datastore service is determined by the storage capacity while the metadata service scales based on simultaneous users and/or request throughput. By offering different services for this functionality, we can scale each according to the application need and separate the performance impact of either heavy data movement activity compared with metadata activity or vice versa.

Second, note that the concept of a file was not chosen as the abstraction for this service. Instead, the relationship between the data storage and the qualities of the metadata we believe need to be separate. This allows use of a custom metadata service optimized for the kind of data organization and structure for a particular application while still using the same data storage service. This choice reflects our belief that future systems will abstract away the file blocking mechanism and instead focus on data sets. This approach has already been successfully demonstrated and adopted by users of the eSiMon [1] simulation dashboard.

4. EVALUATION

To evaluate the scalability and performance of D^2T, tests are performed on Cielo at Sandia National Laboratories. Cielo is a Cray XE6 platform with 8,944 nodes each containing dual, eight core AMD Opteron 6136 processors and 32 GB of RAM. It uses the Cray Gemini network in a 3D torus and uses the Cray Linux Environment. Tests are performed at powers of two scale from 256 up to 65536 processes. We use two pairs of datastore and metadata services along side the process counts mentioned here. Each of these other services is given a single node on which to operate.

The represented IAW consists of a writing application creating ten 3-D variables. Each of these variables is distributed across all of the processes with each process containing a 32x32x32 double precision floating point number chunk for each variable. The total space is scaled by doubling each dimension keeping the total dimensions as close to square as is possible. All ten of these variables and the corresponding chunks are written to the first pair of datastore and metadata servers. Then, an update application marks one variable as "in process" in the first metadata service, reads the chunks in parallel from the first datastore service, updates the data, creates a new variable in a second metadata service inserting the updated data into the second datastore service, and then deletes the original variable from the first datastore and metadata services. For the purposes of this evaluation, only the overhead incurred by the transaction protocol is reported. No attempts were made to make the datastore and metadata service efficient and including those performance results would obfuscate the impact of having transactions surround data movement applications. The overheads are measured during the update application's activities and represent the green and blue arrows within Figure 3.

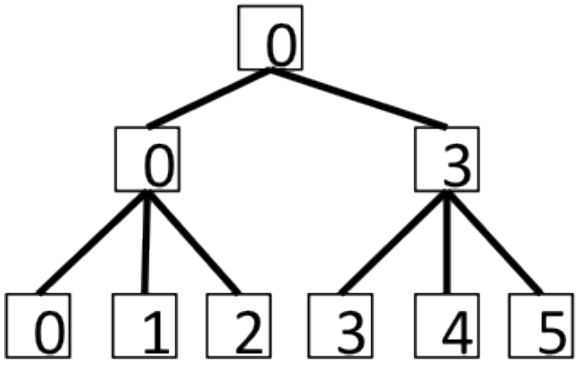

Figure 4: Example Hierarchy

With the three-tier structure employed by D^2T, we choose to always employ at least two sub-coordinators, even at small scale, to incur higher overhead costs. An example of this configuration is shown in Figure 4. In this case, process 0 acts as the coordinator, sub-coordinator, and a subordinate. Process 3 acts as a sub-coordinator and a subordinate. The rest of the processes operate solely as subordinates. The configuration for each run consists of a minimum of 2 sub-coordinators and a maximum of 256 subordinates per sub-coordinator. That yields 256 sub-coordinators each with 256 subordinates at 65536 processes. This is illustrated in Table 1 showing the various scales evaluated and the configuration for each. Other tests of smaller numbers of processes were run, but were omitted because they did not add any additional information to the results.

The test values reported include the following actions:

1. txn_create_transaction - create a new transaction
2. txn_create_sub_transaction - create a new singleton sub-transaction
3. txn_create_sub_transaction_all - called three times to create three global sub-transactions
4. txn_begin_transaction - start the transaction processing
5. txn_commit_sub_transaction - called four times total to commit the four sub-transactions
6. txn_vote_transaction - vote to commit or abort the overall transaction
7. txn_commit_transaction - commit the entire transaction
8. txn_finalize_txn - clean up any transaction specific memory or perform any final operations

All operations except txn_create_sub_transaction and txn_commit_sub_transaction are performed as effectively collective operations. These two exceptions are only performed

by the single process involved. In order for the existence of this singleton sub-transaction to be guaranteed to be known, we split the create and begin operations to provide a time-frame during which all singleton sub-transactions should be initialized. The begin operation gathers knowledge of all of these singleton sub-transactions and broadcasts it to all processes to make failure recovery possible. This step guarantees that every singleton sub-transaction will be known or the begin will fail due to a process failure. Since the global sub-transactions are global operations, they can be done either before or after the begin transaction call. In our case, we have one global sub-transaction created before the begin transaction and two afterwards.

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
8192	32	256
16384	64	256
32768	128	256
65536	256	256

The selected sequence of operations gives a representative idea of what a typical transaction may include. Each test is performed seven times and the mean value is presented. Given the small magnitude of the values, there is a bit of variance that has a large percentage, but small absolute value. The results are presented in Figure 5. The results for 4096 processes were lost.

In the interest of saving space, only the results for 64K processes is discussed. In all cases, the time spent in operations for the 64K case are the longest of all cases tested. At a detailed level, the time for the 64K processes case can be itemized as follows. First, the slowest operation, by far, is the txn_create_sub_transaction_all with a maximum time of 0.0310 seconds for one of the calls. The mean time is 0.01 seconds. Second, the other operations are all at less than 0.005 seconds on average and a maximum of 0.012 seconds. Third, the time for the initialization and cleanup of the total protocol, the calls equivalent to MPI_Init and MPI_Finalize, take 0.38 seconds total. Since the txn_init call is the first call that uses MPI in the system and that it is a full order of magnitude slower than any other operation at worst, we believe there is MPI initial communication channel setup being established accounting for the extra time. The txn_finalize takes a maximum of 0.0002 seconds. Since these calls are done once for the entire application lifetime, we did not include them in the results since they are not part of the typical transaction cost that would be incurred for each transactional operation.

5. DISCUSSION

The leading alternative to D^{2}T is the Intel/Lustre Fast-Forward Epoch approach [2, 10]. While D^{2}T is intended for a general purposes, epochs are designed specifically to address the needs of a shared storage array with a large number of parallel clients. The general idea is to mark every operation with a transaction id, record a process count somewhere to allow reconciliation later, and have every process operate independently for the duration of the writing operation. A second, more synchronous mode is also available. In this synchronous mode, the user is required to manage the transaction ID and to update the transaction state in the storage servers. During a read process, it must be determined what is the most current, complete, "Readable" version of the file. This entails contacting all of the storage devices to determine all of the versions of blocks stored on each device. With that information, it can be determined which of these transactions is "Readable". While this reconciliation operation may happen asynchronously, it must be done before a read operation potentially prompting it to be triggered by a client call or potentially stale data is presented as the most current.

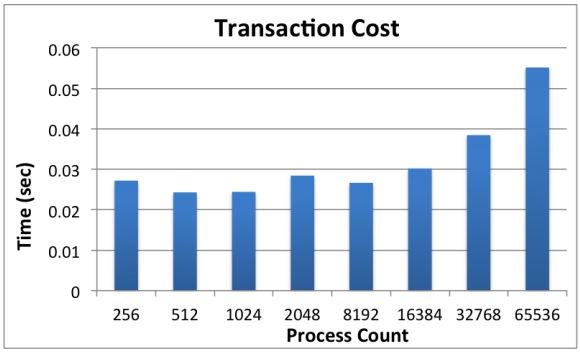

Figure 5: Total Transaction Overhead

Another effort to offer consistency and data integrity for the ZFS file system [11] 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.

6. CONCLUSIONS

We have demonstrated that synchronous two-phase commit transactions can have low overhead and be used with roughly synchronous operations with little additional overhead. While other use cases, particularly those that require a greater degree of asynchrony, require a different approach, D^{2}T offers a working solution for a wide variety of scenarios today. The bigger question of what sort of synchronization mechanism is appropriate for various use cases is currently under investigation.

The evaluation of the fault detection and recovery are beyond scope for this paper and are in preparation for presentation elsewhere.

7. 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.

8. REFERENCES

[1] R. Barreto, S. Klasky, N. Podhorszki, P. Mouallem, and M. A. Vouk. Collaboration portal for petascale simulations. In W. K. McQuay and W. W. Smari, editors, *CTS*, pages 384–393. IEEE, 2009.

[2] E. Barton. Lustre* - fast forward to exascale. Lustre User Group Summit 2013, March 2013.

[3] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In B. N. Bershad and J. C. Mogul, editors, *OSDI*, pages 335–350. USENIX Association, 2006.

[4] 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.

[5] 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.

[6] 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.

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

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

[9] 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.

[10] J. Lombardi. High level design - epoch recovery, june 25th, 2013. Intel FastForward Wiki, June 2013.

[11] 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.