Efficient, Failure Resilient Transactions for Parallel and Distributed Computing

Jay Lofstead*, Jai Dayal†, Ivo Jimenez‡, Carlos Maltzahn‡
*Sandia National Laboratories gflofst@sandia.gov
†Georgia Tech jdayal3@cc.gatech.edu
‡University of California, Santa Cruz {ivo@cs,carlosm@soe}.ucsc.edu

***Abstract*—Scientific simulations are moving away from using centralized persistent storage for intermediate data between workflow steps towards an all online model. This shift is motivated by the relatively slow IO bandwidth growth compared with compute speed increases. The challenges presented by this shift to Integrated Application Workflows are motivated by the loss of persistent storage semantics for node-to-node communication. One step towards addressing this semantics gap is using transactions to logically delineate a data set from 100,000s of processes to 1000s of servers as an atomic unit.**

Our previously demonstrated Doubly Distributed Transactions (D^2T) protocol showed a high-performance solution, but had not explored how to detect and recover from faults. Instead, the focus was on demonstrating high-performance typical case performance. The research presented here addresses fault detection and recovery based on the enhanced protocol design. The total overhead for a full transaction with multiple operations at 65,536 processes is on average 0.055 seconds. Fault detection and recovery mechanisms demonstrate similar performance to the success case with only the addition of appropriate timeouts for the system. This paper explores the challenges in designing a recoverable protocol for doubly distributed transactions, particularly for parallel computing environments.

I. Introduction

Integrated Application Workflows (IAWs) are projected to be essential for scientific computing for extreme scale platforms. These IAWs integrate scientific simulations and data analytics tools into a more tightly connected workflow. Key with this model is the move to eliminate using centralized storage for intermediate data storage between the workflow components. Complicating the picture is the growing imbalance between compute capacity and IO bandwidth. Compound this with the limited in compute area storage prompting rapid processing to avoid bottlenecks. Additional pressure from the energy cost for moving data is also a concern. The current standard of using the persistent storage system for staging intermediate results becomes doubly infeasible. Instead, the focus is on developing a replacement mechanism that replaces the centralized persistent storage mechanism with in compute area data storage and integrated processing. Unfortunately, the additional complexity of having parallel clients all generating part of the atomic message to a parallel collection of servers each storing only a portion of the whole message eliminates the possibility of using existing protocols like Paxos or Zookeeper.

Many challenges must be addressed before IAWs can replace persistent-storage based workflows. First is addressing the semantics offered by persistent storage, such as encapsulating data into sets frequently represented by individual files or file sets. The current techniques delineating individual data sets for persistent storage devices are problematic at best. While the file abstraction works well for data set delineation, when that set is complete and if it is correct is a continuing problem. With varying degrees of success, many techniques have been employed for detecting when a data set is complete. For example, looking for a guard file or when a file stops changing size inadequately address the entire storage stack. The caching at various layers may delay the actual data persistence in spite of these markers being met. Some workflow systems address this by delaying an entire output cycle trusting that when the next output starts, the previous one has been fully flushed and is available for processing. These sorts of delays ultimately slow down the scientific discovery process. Ideally, a mechanism that managed a data output would also address this completeness discovery instead of just the data set encapsulation.

Determining when an operation is complete and correct is a general problem for dynamic IAWs. The most effective use of available compute resources will require resource reallocations over time not just to compensate for a sub-optimal initial allocation, but also to address failures and changes in the applications causing shifts in the volume and/or frequency of data or even the kind of processing that should occur. A prime example of this kind of processing are Adaptive Mesh Refinement codes. The initial running state is a coarse grid that has refinements in "interesting" areas generating more intensive computation and much larger data volumes. To properly address these events without unduly affecting the various components, resources should be properly deployed, configured, and running prior to any upstream component use. Should a failure occur in this reallocation process, a different decision on how to deploy resources can be tried without causing an upstream error beyond the currently detected bottleneck. Ideally, any solution that can handle the data movement operations could also be deployed to handle these dynamic system reconfiguration operations.

Our initial work to address these scenarios yielded the D^2T protocol [14]. It demonstrated how to create a protocol to manage data movement and system reconfiguration operations. While the initial version worked, it was only a somewhat scalable MxN two-phase commit protocol. First, the server side was required to manage transactions directly forcing inclusion of both a communication mechanism between clients and servers for control messages and 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 address scalability. The aggregate size and/or quantity of messages

978-1-4673-6750-9/14 $31.00 © 2014 IEEE
DOI 10.1109/DISCS.2014.13

IEEE computer society

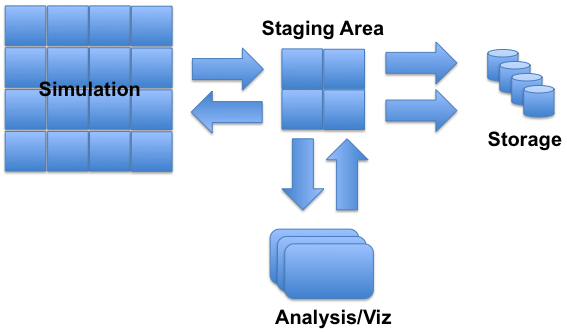


Fig. 1. IAW Model

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. While this is much less of an issue for dynamic system reconfiguration operations, it is likely deadly for data movement operations at scale. To address these limitations, we redesigned the protocol and incorporated failure detection and recovery.

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 unit is more straightforward. Existing file systems use the file concept with locks controlling reading and writing, depending on the user and the operation, to achieve this semantic. This works well 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, data set pieces 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.

The kind of IAW environment we are targeting is illustrated in Figure 1. Transactions manage data movement from the simulation to the staging area, the data processing in the analysis/visualization routine, and the subsequent data movement to storage. In addition, analysis/visualization area management is considered. The transaction protocol helps manage resizing the resource allocation of the various analysis and visualization components. This has been described and analyzed previously [5]. This additional use for D^{2}T demonstrates its generality. For this evaluation, we focus on the data movement case because it is the most intensive and demonstrates the possibilities at scale.

The contributions of this paper are as follows:

- an exploration of fault detection and recovery mechanism requirements for such a protocol
- a discussion of the requirements for incorporating D^{2}T into or as wrappers for existing services

Through these contributions, IAWs are one step closer to providing the functionality and guarantees application scientists require for adopting that workflow model.

The rest of this paper is organized as follows. First, related work is discussed in Section II. Next, the protocol design is presented in Section III. Section IV discusses how to handle a failure for each of the different process roles. Next, demonstration support tools and lessons for incorporating transaction support into systems are discussed in Section V. The evaluation is presented in Section VI. Finally, conclusions are presented in Section VII.

II. Related Work

Alternatives, such as Paxos [11] algorithms like ZooKeeper [7], 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 insufficient to hold 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 require, 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 leading alternative to D^{2}T is the Intel/Lustre Fast-Forward Transaction/Epoch approach [2], [16]. While D^{2}T is intended for 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, 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 must manage the transaction ID and update the transaction state in the storage servers. During a read process, the most current, complete, "Readable" version of the file must be determined. D^{2}T has no dependence on earlier versions.

Another effort to offer consistency and data integrity for the ZFS file system [17] 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 data movement integrity in addition to agreement about the successful completion of a parallel operation.

Chandra and Toueg propose a taxonomy where eight classes of failure detectors for asynchronous systems are specified and describe completeness and accuracy properties for each [4]. They also show how to solve the consensus problem for each class. In this work we assume a synchronous network by making use of a time-out based failure detection mechanism and implement two-phase commit [6] on top of it to achieve consensus.

The MPI-3 fault tolerance working group has proposed adding a new MPI_Comm_validate_all directive to the MPI standard that collectively and reliably returns a list of failed processes. Our work relates in that a consensus protocol is implemented. The difference is that our agreement protocol is demonstrated on top of MPI rather than internally. This

eliminates any dependence on MPI features and enables cross-application transactions. A tree-structured implementation of two-phase commit [6] similar to the one presented here was introduced in [8]. A dynamic protocol subsequently appeared in [3]. Both of these approaches abstract an external failure detection mechanism on which their protocol relies on. D^2T provides the failure detection mechanism.

Failure detection for MPI has been previously researched. In [9] the authors propose a framework supporting several types of failures. The framework runs within a particular MPI implementation. In [10] two timeout-based failure detection approaches that run on top of MPI are introduced. The first one uses a "liveness" check that adaptively chooses when to trigger failure detection, while the second periodically checks for failed nodes. Consensus is treated orthogonally in these.

III. Protocol Design

The new protocol was presented in detail previously [13]. For clarity with the failure detection and recovery semantics, it will be summarized here.

The original protocol had the clients and servers each manage part of the transaction requirements. Servers and clients were required to coordinate with their peer group of servers or clients respectively to handle parallel semantics. The revised protocol design removes the requirements for server processes and assumes that clients can acknowledge the success or failure of a particular server process. Wrappers or interfaces around server processes, as discussed in Section V, can be used to replace the missing semantics for existing services. The overhead for these wrappers is also discussed.

To address client-side scalability, processes are organized into a hierarchy with an overall coordinator, a collection of sub-coordinators, and subordinate processes. How this works is discussed in more detail elsewhere [13] as well as briefly as part of the discussion of fault detection and recovery. The performance evaluation has been expanded and is included in Section VI.

IV. Failure Detection and Recovery

We proposed using timeouts for failure detection and recovery previously [14], but the idea was never fully thought through nor implemented. It has since been rethought and implemented to work within the hierarchical structure in the current implementation.

The basic mechanism still depends on timeouts to determine that a process is no longer active or reachable. Depending on the role of a process, the steps for recovery are different. As an overall description, some observations are made and drive the requirements for all of the different roles.

First, for failure recovery to work, any process must be able to determine what the overall system status was and potentially take over any role. For example, if a sub-coordinator fails, any of its subordinates must be able to step into the role for the system to effectively continue. If the overall coordinator fails, the same holds true. This prompts the requirement that any failure be globally known for the system to be able to recover. Further, the full list of singleton and global transactions must be known by all processes or they cannot step into a role assigned to them. This requirement does incur additional messaging and storage requirements, but there is no other way to guarantee that continued operation or recovery may be possible. This additional messaging has been incorporated in the protocol.

Second, a suitably long timeout must be selected to not cause false positives without introducing undue delays when a failure is detected. Fortunately, the typical and even worst case observed performance is just a fraction of a second. There is an additional challenge introduced by the hierarchical structure of the new protocol. In the case of a subordinate failure, the sub-coordinator must wait for the timeout interval. The difficulty is that the coordinator is also waiting on the sub-coordinator. That prompts the need for a longer timeout period. The specifics of these periods and how they are handled for each role is detailed below.

The current three-tier implementation uses two values for timeouts. The first timeout value is called *SHORT*. This timeout is used as the general case. The second one we call *LONG* we default to twice as long as SHORT. The actual length required for LONG is probably less than this value. Consider having a subordinate fail. The sub-coordinator must wait for the SHORT time to determine that the process is no longer available. Then the sub-coordinator must notify around about the failure. The LONG used by the coordinator in this case represents the SHORT time plus the notification time. The specific increment beyond SHORT is extremely dependent on the scale and the interconnect. For a sufficiently fast interconnect, this value could be a small increment beyond SHORT. For cases where communication delays may be either inherent or possible, this longer timeout would need to be much longer. The basic test of using twice the SHORT is solely for convenience rather than trying to determine an optimal increment for different interconnects, scenarios, and scales. This sort of evaluation is left for future work.

While this timeout idea works at a basic level, it is insufficient to handle all of the failure cases. The specifics of how these timeouts are adjusted are discussed in each case below. The basic tree structure messaging uses the initial timeouts listed in Table I to detect when a failure has occurred:

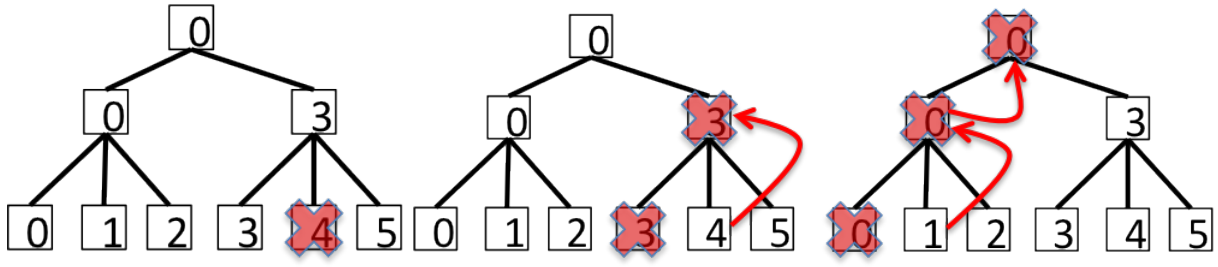
TABLE I. Messaging Timeout Values for Each Role

Phase	Role	Operation	Timeout
gather	subordinate	send to sub-coordinator	SHORT
	sub-coordinator	receive from subordinates	SHORT
		send to coordinator	SHORT
	coordinator	receive from subordinates	SHORT
		receive from sub-coordinators	LONG
broadcast	coordinator	send to sub-coordinators	SHORT
		send to subordinates	SHORT
	sub-coordinator	receive from coordinator	SHORT
		send to subordinates	SHORT
	subordinate	receive from sub-coordinator	LONG

For the gather process, messages are collected from the subordinates at the bottom of the tree up to the coordinator. Sufficient time for the sub-coordinators to react to a failure without the coordinator assuming the sub-coordinator has failed is required prompting the LONG timeout.

For the broadcast process, the opposite is true. The coordinator waits for the shorter wait period for the sub-coordinators to receive the message. Then the subordinates need to wait for the LONG wait period before assuming a failure has occurred.

While these timeouts are fine for detecting when a failure occurs, processing that failure and resuming "correct" operation is more complicated as explained below.



(a) Subordinate Failure Process (b) Sub-Coordinator Failure Process (c) Coordinator Failure Process

Fig. 2. Failure Processes

A. *Subordinate Failures*

Subordinate failures are the most straightforward because they largely stay localized. In Figure 2(a), process 4 will be used to illustrate the process.

Using the standard timeout mechanism, process 3 detects that 4 has not responded to a message. At this point, process 3 notifies the other subordinates, just 5 in this case, and the other sub-coordinators, just 0 in this case. Next, all of the sub-coordinators tell all of their subordinates that process 4 has failed.

To address the timeouts properly, process 3 uses a SHORT timeout value to detect the failure. Then, process 0, in the role of the coordinator, is waiting for process 3 to respond. It waits for the LONG timeout period.

In this case, the simple timeout values are sufficient for proper operation.

B. *Sub-Coordinator Failures*

When a sub-coordinator fails, both the subordinates that are managed by it and the overall coordinator have to be managed. Consider the failure of process 3 in Figure 2(b).

In this case, the failure is detected in multiple locations simultaneously. First, all of the subordinates of process 3, processes 4 and 5 in this case, notice that 3 is not responding as expected within the SHORT timeout period. At this point, the subordinates must decide among themselves which process should assume the role of sub-coordinator. The current mechanism is to use the lowest rank numbered process still valid as the new sub-coordinator. All of the process in this group can come to this conclusion independently because we use a standard formula to determine what role a process plays initially and have distributed global failure knowledge. The new sub coordinator, 4 in this case, tells its peers that process 3 has failed. Just like with the subordinate failure, this notification is sent to all of the subordinates. In this case, because a sub-coordinator failed, the aggregation step never occurred for that sub-tree and the overall coordinator causes the operation overall to fail. It could be retried with the new tree assuming nothing unrecoverable was lost. Just as with the subordinate failure case, the coordinator timeout offers sufficient time for the detection and notification propagation of the failure to all processes without causing a timeout due to the additional messaging and processing.

The complication comes in what processes 1 and 2 do because of the timeout. In this scenario, when the gather operation fails, process 0 has already collected the data from 1 and 2, in the role of a sub-coordinator, Processes 1 and 2 are then waiting for the LONG time for the response. The problem is that process 0 is timing out using the LONG time when waiting for process 3. The problem is that processes 1 and 2 are already waiting for the broadcast from 0 using the LONG time. Before they are notified with the broadcast, they need to have their timers extended to account for the extra time spent with the detection and recovery of the failure of process 3. To accomplish this, when a process receives a message about a process failure, the timer is extended for another LONG time to ensure there is sufficient time for processing of a failure without falsely detecting a second failure. Investigation of multi-failure cases is left for future work and may require some adjustment to this algorithm.

C. *Coordinator Failures*

When the overall coordinator fails, only the sub-coordinators need to worry. However, it is not quite that straightforward. In Figure 2(c), process 0 represents the overall coordinator.

In this case, the subordinates have all sent out their messages to their sub-coordinators and are waiting for the response back. In this case though, processes 1 and 2 treat the coordinator as a sub-coordinator as well with a SHORT timeout. The problem is that the sub-coordinators also are waiting with a SHORT timeout on the coordinator to accept their aggregated messages. In this case they need to abort their wait and tell their subordinates to abort waiting as well. If they continue to wait, there will not be any possible downward message broadcast possible other than to indicate the failure. In this case, the wait can be short-circuited avoiding any additional wait times. The messaging about the coordinator rank failure triggers this short circuit.

D. *Discussion*

The basic mechanism of timeouts, with some adjustments for different failure cases, works well for simple failure cases. It is capable of handling a failure at different levels of the hierarchy without introducing false positives for failures. In the case of multiple failures, the process may need to change. This scenario has not been considered and is left for future work. The case of a failure during the gather and broadcast process has not been fully explored either. These cases to fully explore the full set of possibilities for failures are left to future work.

V. Demonstration Support Tools and Lessons

To better understand the requirements for using D^2T within a data staging scenario, the basic functionality of a datastore and a metadata service were created. These services are not

intended to be performant nor scalable, but instead to help learn about the minimum requirements that can be imposed on the server side of a doubly distributed transaction. They are created as separate services to better isolate the specific requirements of the different functionality. An IAW may or may not require a metadata service to describe the data shipped between components. Instead, a known data storage service and direct communication of object IDs could be used. IAWs may alternatively perform direct data movement based on a request requiring a metadata service without a datastore service. Because either of these services may be optional, it is clearer to determine the minimum required functionality separately. The specifics of the special features required of these two services are described briefly below to help support the efficacy of the D^2T approach for safely managing data movement and other atomic operations.

A. 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 based on some unique identifier. The actual data storage location may vary. For example, it may be physically on one of the same nodes as the application. This offers the advantage of reducing data movement at a cost of exposing the data to potential node faults. Beyond this basic functionality, the datastore service requires a small additional functionality increment.

As an additional part of the store and retrieve calls, a transaction ID can be provided. This affords filtering object visibility based on transaction membership. Internally, in addition to the transaction ID, a state is required. This state attribute may be an extension of an existing attribute used to mark an object as deleted rather than a new requirement. In this case, we extend the attribute to have a few new states. First, an "in process" state marks an object as only visible to members of the same transaction. Second, an "active" state says that the object is available to any client.

From an API perspective, there are a few requirements. First, the object manipulation calls must be extended to include a transaction ID. Second, additional calls that can be used to clean up failed transactions and to mark objects part of a successful transaction are required. While other functionality, such as authorization controls, are necessary for a production system, this demonstration offers no additional functionality.

By isolating this functionality and making a minimal extension, we demonstrated that it is possible to add a thin wrapper around an existing data storage service to handle these additional requirements by offering new calls and an internal cache of object IDs, associated transaction IDs, and object states to represent the required extended functionality.

B. Metadata Service

The requirements for a metadata service along side the datastore service described above to be generally useful has been presented previously [12]. A summary of some of the details as well as the required extensions to support D^2T are provided below. For the above publication, the implementation was incomplete, but the functionality was thought through. Since then, the implementation has been completed and proved in the demonstration system. Many details have been determined fleshing out the requirements in more detail.

The basic system is designed around storing and retrieving objects representing data from a single process for a single variable. For example, when ADIOS [15] writes out data, it creates a log-based format with each process writing out a blob with sufficient annotation to reconstruct a logical global view later. This minimal coordination approach is likely required for extreme scale IAWs. 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 may be required for a general system. Using the rectangular simulation domain, the metadata service 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. 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.

To extend this service to support transactions, a few changes are required. First, a transaction ID is required to mark a variable as part of a particular transaction controlling visibility. Second, a state attribute must be extended to support the "in process" and "active" states.

Just as with the datastore service, a thin wrapper can be made for an existing metadata service to provided the additional functionality required to support transactions. Unfortunately, the space requirements are different from the datastore service. For the datastore service, the cached data to extend the service to support transactions is very small compared to the datastore service requirements. In the case of the metadata service, the cached data is a significant fraction of the data storage requirements of the metadata service itself. That makes such a wrapper a more significant impact that requires more careful consideration.

C. 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.

Third, these demonstration systems have shown that a small increment beyond the core functionality is required beyond the basic functionality of these services. In the case of the datastore service, a small wrapper can be provided around an existing service with small space/time impact. For the metadata service, the impact is much more significant. Just from a space perspective, approaching twice as much space is required to cache sufficient information to wrap the metadata to add the transaction requirements.

VI. Evaluation

The evaluation is split into two parts. The first part focuses on the overhead of the protocol in the normal operating scenario of no failures. The second part examines the overhead of detecting a failure and returning control back to the client code.

A. General Performance Evaluation

To evaluate the scalability and performance of D^{2}T, 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.

With the three-tier structure employed by D^{2}T, 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 2. 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 II 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 timeframe 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 II. 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

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 3.

In the interest of saving space and avoiding including multiple nearly identical graphs, 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.

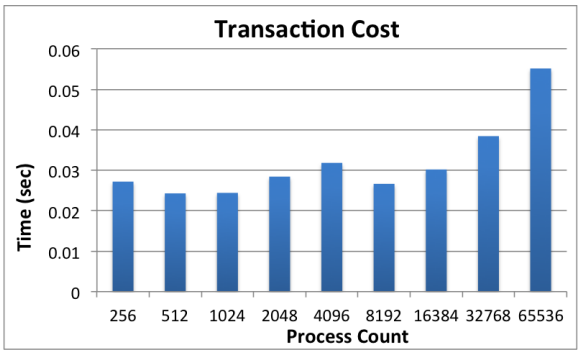


Fig. 3. Mean Total Transaction Overhead

B. Failure Detection/Recovery Evaluation

To test the failure recovery process, we inserted code for a single process to skip the rest of the test harness to represent a failure. Otherwise, the tests are identical to those performed for the general evaluation.

Unlike for the success cases, the results presented here represent the worst case times for all of the tests. We chose this metric because the impact of the worst case is critical for determining if fault detection and recovery is too expensive. Also of general note is that SHORT is set to 5 seconds and LONG is set to 10 seconds. As the results below in conjunction with the success values presented above demonstrate, these values can be greatly reduced without impacting the viability of the approach. Further evaluation to determine what the correct values should be for a given configuration is beyond the scope of this paper.

1) Subordinate Failure Overhead: For a subordinate failure, as illustrated in Figure 2(a), the failure is localized to a particular sub-tree. In this case, this represents the overhead to detect and reach a state where a failure has been detected and the entire set of participants are both aware of the failure and have adjusted accordingly. The times are presented in Figure 4(a). While the 256 process count appears to take a significantly longer time (5.15 seconds vs. 5 seconds for 512 processes), the difference is actually about 3%. This was due to recreating these results at a later time with different system activity. It is also worth pointing out that the difference between 512 processes and 65536 processes is about 6% or 0.3 seconds. By taking away the 5 seconds for SHORT, the time for the messaging is at most about 0.3 seconds worst case. That would make a SHORT time of around 0.5 seconds likely sufficient.

2) Sub-Coordinator Failure Overhead: For the sub-coordinator failure example illustrated in Figure 2(b), the process is a bit more complex. In this case, when process 3 is lost, process 4 must be promoted to take over the sub-coordinator role. This illustrates the importance of global knowledge of failures for proper system operation. When the coordinator needs to send messages down the tree, it needs to know that process 4 is the new sub-coordinator for this tree. By using the calculation approach to membership and succession order, special messages for new roles are unnecessary. The simple process failure message is sufficient. The performance for failure detection is presented in Figure 4(b).

In this case, the time difference between 256 processes and 65536 is even smaller at 0.034 seconds. The 10 seconds for the LONG again is radically large and can be cut down considerably.

3) Coordinator Failure Overhead: The coordinator failure case is illustrated in Figure 2(c). In this case, process 0 has three roles requiring an adjustment both at the coordinator and sub-coordinator levels. Further evidence of the importance of global knowledge of the system state is illustrated here. In this case, process 1 must be able to take over the overall coordination in addition to the sub-coordinator role. Again, the processes are all notified about the failed rank and use local calculation to determine which process will step into the new role(s). Figure 4(c) illustrates the performance overhead. Similar to the sub-coordinator failure case, the overhead difference between the 256 and 65536 processes case is about 0.045 seconds. This again a demonstration of how small the wait times can be for the timeout, at least for Cielo, and still offer a functional system with few, if any, false positives for failure detection.

C. Discussion

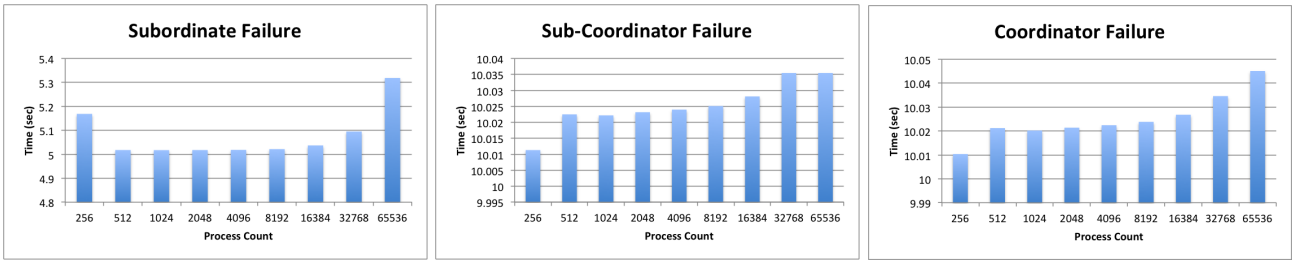
While the performance is excellent both for the general case and for the failure detection and recovery cases, these numbers are not as good as they could be in an ideal situation. The default implementation of D^2T uses an MPI transport out of implementation simplicity. The simulation of a failure by having a process skip the rest of the test harness does not block the default implementation of MPI_Isend's behavior (for the Cray MPI implementation). The default behavior is to use a buffered send meaning that as long as there is buffer space available on the destination node, the message is considered sent. For the general case, this is fine and helps make an application less synchronous. In our case, we needed to ensure that the message was not received at the destination node. This required shifting to using MPI_Issend instead. This synchronous send version requires that the message be received before the send asynchronous send call is considered successfully completed. This effectively slows down the implementation. The implication is that in the case of an MPI implementation that is fault tolerant, D^2T could switch to using the buffered send instead slightly improving the performance.

The need for a truly asynchronous messaging setup for more asynchronous application structures will require rethinking the fault detection mechanism. This is left for future work.

VII. Conclusions and Future Work

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^2T 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 fault detection mechanism of using timeouts has proved effective, even at scales representative of extreme scale platforms. The amount of time allocated for the timeout can be lowered, but at the risk of causing a false positive. The



(a) Subordinate Overhead (b) Sub-Coordinator Overhead (c) Coordinator Overhead

Fig. 4. Worse Case Failure/Recovery Overhead

real challenge for this portion of the protocol to be useful is the development of an effective approach for application messaging, such as MPI, to survive a process fault with temporarily degraded performance while the work is absorbed elsewhere.

The proper structure of the aggregation tree for optimal performance and scalability has not been explored. Various different structures can be evaluated to determine which structure will yield the best results under different conditions.

Another case not investigated is when the aggregate size of all of the singleton and global sub-transactions grows so large that they cannot be stored within a single process effectively. While this is unlikely in current applications, the shrinking memory per process as machines scale may yield a sufficiently constrained environment that this becomes an issue. In this case, some federation of trees could be deployed to replace the single overall coordinator structure in use today. The performance and reliability implications of this would require additional thought and investigation.

An effective framework for determining the proper time-out values in order to reduce possible delays needs to be investigated still. The networking community has certainly done some pieces of this work, but it would need to be expanded to consider things like the number and sizes of messages, the number of processes aggregating into a single point, how busy the target platform is, and potentially other factors. Determining an effective approach and what the set of factors to consider is a project in itself.

The promised comparison against the asynchronous, storage focused epochs approach of the Intel/Lustre FastForward project is underway and being prepared for submission elsewhere.

VIII. 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.

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] D. Buntinas. Scalable distributed consensus to support MPI fault tolerance. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 1240–1249, 2012.

[4] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4):685722, July 1996.

[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] J. Gray. Notes on database operating system: An advanced course lecture notes in computer science. *Springer Verlag, Berlin*, 1(60):393–481, 1978.

[7] 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.

[8] J. Hursey, T. Naughton, G. Vallee, and R. L. Graham. A log-scaling fault tolerant agreement algorithm for a fault tolerant MPI. In Y. Cotronis, A. Danalis, D. S. Nikolopoulos, and J. Dongarra, editors, *Recent Advances in the Message Passing Interface*, number 6960 in Lecture Notes in Computer Science, pages 255–263. Springer Berlin Heidelberg, Jan. 2011.

[9] H. Jitsumoto, T. Endo, and S. Matsuoka. ABARIS: an adaptable fault Detection/Recovery component framework for MPIs. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8, 2007.

[10] K. Kharbas, D. Kim, T. Hoefler, and F. Mueller. Assessing HPC failure detectors for MPI jobs. In *2012 20th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 81–88, 2012.

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

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

[13] 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.

[14] 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.

[15] J. Lofstead, F. Zheng, S. Klasky, and K. Schwan. Adaptable, metadata rich IO methods for portable high performance IO. In *Proceedings of the International Parallel and Distributed Processing Symposium*, Rome, Italy, 2009.

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

[17] 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.