D^{2}T: Doubly Distributed Transactions for High Performance and Distributed Computing

Jay Lofstead[1], Jai Dayal[2], Karsten Schwan[2], Ron Oldfield[1]
[1]CSRI, Sandia National Labs.
[2]CERCS, Georgia Institute of Technology

***Abstract*—Current exascale computing projections suggest rather than a monolithic simulation running for the majority of the machine, a collection of components comprising the scientific discovery process will be employed in an online workflow. This move to an online workflow scenario requires knowledge that inter-step operations are completed and correct before the next phase begins. Further, dynamic load balancing or fault tolerance techniques may dynamically deploy or redeploy resources for optimal use of computing resources. These newly configured resources should only be used if they are successfully deployed.**

Our D^{2}T system offers a mechanism to support these kinds of operations by providing database-like transactions with distributed servers and clients. Ultimately, with adequate hardware support, full ACID compliance is possible for the transactions. To prove the viability of this approach, we show that the D^{2}T protocol has less than 1.2 seconds of overhead using 4096 clients and 32 servers with good scaling characteristics using this initial prototype implementation.

I. Introduction

As scientific application scale towards exascale, they will incorporate more complex models that have previously only been run as separate applications. For example, in fusion science, simulation of the edge of the plasma [1] and the interior of the plasma [2] are currently separate. To have a more complete, accurate model for a fusion reactor, these components will need to be tightly coupled to share the effects between the two models. The CESM climate model [3] is similar in that it incorporates atmosphere, ocean, land surface, sea ice, and land ice through a coupling engine to manage the interactions between each of these different systems yielding a more accurate model of global climate. In most cases, these and other scientific applications are part of larger offline workflows that process the output written to storage in phases that ultimately yields insights into the phenomena being studied. Current work to enable these coupling and workflow scenarios are all focused on the data issues to resolve resolution and mesh mismatches, time scale mismatches, and simply making data available through data staging techniques [4]–[9]. In most of these cases, each of the components are run using a separate execution space for fault isolation and to aid in scalability.

In a similar vein, fault tolerance techniques must evolve to offer more application stability in the face of faults without relying solely on the storage system to store checkpoints. The ever increasing ratio of compute speed to IO bandwidth demands rethinking resilience to avoid using centralized disk-based storage for periodic checkpoints. For example, dynamic redeployment of new resources to replace failed components could enable moving the application to a correct state to enable continuing processing without having to redeploy the code, read a checkpoint, and start processing again. Similar to deploying resources for fault tolerance, load balancing and system reconfiguration tasks [10], [11] support dynamically varying the amount of resources dedicated to different components based on the application needs, machine performance characteristics, and resources available. For both the code coupling/online workflows cases and the fault tolerance/dynamic reconfiguration scenarios, a key feature is missing.

For an operation to proceed that depends on external resources, such as a coupled code waiting for external data or waiting for deployment of additional resources for fault tolerance or load balancing, some mechanism to know that the operation is both complete and correct is necessary. We propose adopting a transaction mechanism similar to what is offered in databases, but adapted to the highly distributed environment of parallel computing. The database community has solved how to perform distributed transactions, but with a limitation that makes them insufficient for parallel computing: 1 client to N servers precludes having a collection of clients performing a collective operation to participate in a single transaction. For the HPC environment, particularly for these more general scenarios, this is too limiting to be useful. This paper extends the idea of distributed transaction such that they are doubly distributed, i.e., distributed on both the client and server sides, dubbed D^{2}T.

The D^{2}T protocol aims to offer full ACID-style guarantees for the encapsulated operations. While this is certainly possible with adequate hardware particularly to support the durability guarantee, this initial work shows that such a system can be built that supports most of the ACID-style guarantees with current hardware, what the performance impact will be, application coding implications, and begin to address the scalability challenges so that it is applicable for exascale-sized platforms. More specifically, D^{2}T can address both the code coupling/online workflow/data staging as well as the fault tolerance/system reconfiguration scenarios.

Through the D^{2}T protocol, we enable data staging to move from hiding IO costs or to performing some "in-flight" processing into a way to move offline workflows into online workflows that eliminate, or at least greatly reduce, the use of slow, centralized, persistent storage resources while addressing much of the functionality loss of losing the persistent storage

intermediate. Further, it offers a standardized mechanism for fault tolerance/system reconfiguration tasks to prevent premature or incorrect use of resources by the existing system. Since the goal of ensuring an operation is both complete and correct is shared between both scenarios, this single protocol can be used for both simplifying programming.

Our initial approach aims for a complete, rather than optimal protocol. Ongoing efforts are identifying which messaging can be eliminated or combined without loss of guarantees as well as other optimizations. This initial approach adds a few extra rounds of messages during an operation to coordinate the transaction state. By inserting a small amount of additional metadata in these messages, we are able to track the state of the transaction and related operation triggering appropriate actions at the correct times.

The remainder of this paper is organized as follows. Section II presents a short overview of the related work. We introduce the D^{2}T protocol in Section III. We next present our initial implementation in Section IV. This includes a description of the failure modes we are considering and some general information about how they are addressed. Section V describes our validation experiments and the generated results while Section VI presents our conclusion.

II. Related Work

The concept of distributed transactions [12] has been around for decades. By developing a protocol for managing a collection of distributed resources for an atomic operation has offered tremendous benefits for scalability and the diversity of applications using the technology. The extension of this technology to address distributed clients working in concert, while not critical for the core database area, is crucial for HPC applications given the massively parallel nature of the modern HPC environment.

ZooKeeper [13] and other Paxos [14] implementations have a superficial similarity to our D^{2}T protocol in that they provide the consistency and synchronization mechanisms for messaging to a collection of servers from a distributed set of sources. Under the hood, Paxos is solely 1xN with an eventual consistency model. The inherent assumption that an update or insert originates from a single source limits the applicability of the protocol for this environment.

GridFTP [15] extends traditional FTP to provide reliable high-performance, parallel data movement in a grid computing environment. Beyond the limitation of GridFTP to data movement, D^{2}T is designed to be used in a time-critical environment rather than before or after the simulation run. D^{2}T is also designed to scale to potentially millions of cores on one side communication with thousands on the other and protect users from incomplete or incorrect data.

Sinfonia [16] is designed so that nodes can share data scalably and fault-tolerant while shielding the developer from the message passing paradigm. To do this, Sinfonia defines a "mini-transaction" primitive that allows applications to manipulate distributed state in a consistent and fault-tolerant manner. Sinfonia's notion of a distributed transaction differs from ours in that Sinfonia employs 1xN semantics where as we need MxN semantics.

Yahoo!'s PNUTS [17] is a large-scale distributed data store designed to run in geographically distinct locations that organizes the data store into ordered tables and aims to provide per-record consistency guarantees. This work lacks the MxN semantics required for our scenarios and uses an eventual consistency model inappropriate for the HPC environment.

While Cassandra [18] and G-Store [19] have provided novel contributions for their intended use cases, they fall short for our needs in a few ways. First, distributed transactions for these systems employ traditional 1xN semantics where as we require MxN semantics. Second, the eventual consistency model will not work for the HPC environment as allowing analysis or visualization tools to operate on stale data is useless and expensive. Further, given the limited resources available in the computation area, updates must be made quickly to reduce the memory requirements for any staging area.

Not mentioned, but fairly universal in these and related tools is the use of a disk-based log. While this likely could be removed in these systems with a relaxed level of durability, it is currently not an optional feature of these systems.

III. The D^{2}T Protocol

The D^{2}T protocol provides the necessary extensions to traditional distributed transactions to afford extension to distributed clients as well as distributed servers, hence the doubly distributed transactions. Figure 1 presents an abstract model representing how this protocol works.

To handle a collection of operations as a single transaction in such a highly parallel environment, a simple, single level transaction is inadequate because we need to spend some effort to manage the set of clients and servers to ensure that each operation is complete and correct. D^{2}T provides a two-level system where a master transaction represents the entire collection of operations comprising the atomic whole and the sub-transactions represent the individual component operations that should be handled as a unit. In Figure 1, the color of the arrows categorize what that logical message represents. The master transaction messages are represented in the pink color, the sub-transaction messages are green, while the actual operation being wrapped in the sub-transaction is blue. It is expected there will be a series of sub-transactions represented by the green-blue-green series of messages/events in the figure.

A. Logical Protocol

Logically, the client side coordinates to agree upon the transaction parameters such as an ID and which client(s) and server(s) are part of the overall transaction. This information is communicated to the server side initializing any expectations such as buffers and identifying the group of servers participating in this collection of operations. Once the servers have initialized this transaction, they respond that they are ready. Next, the clients perform a series of sub-transactions. This affords the opportunity to isolate a single action and offers an opportunity to retry failures before giving up entirely. Finally, assuming everything has worked properly up to this point, the clients coordinate with the servers to commit the operation(s) and close the transaction. This will allow cleaning up any

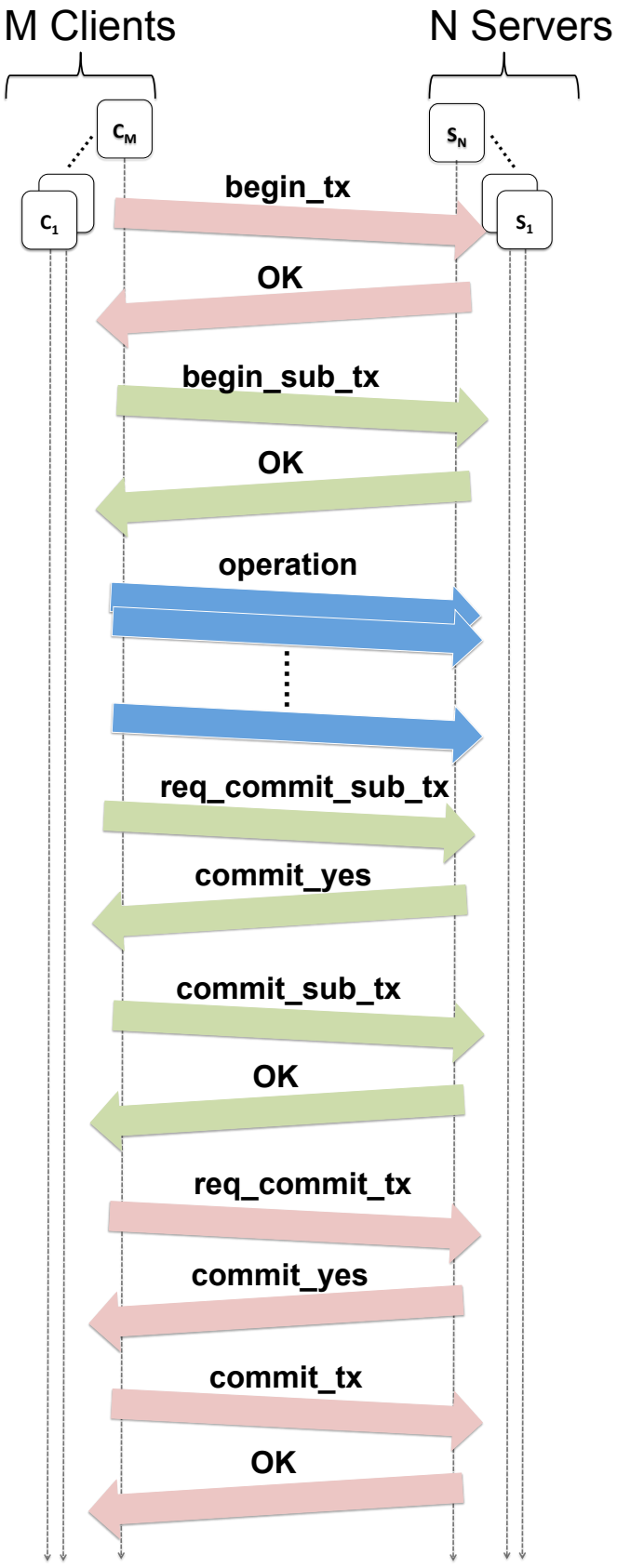

Fig. 1. Doubly Distributed Transaction (D^2T) Protocol

buffers, performing any cleanup operations such as deallocating replaced components, making new resources available to clients, or cleaning up other resources allocated to handle this transaction.

The logical protocol is described in more detail below giving details and explanation for why each piece and necessary and how it contributes to the overall goals of D^2T.

1) Initialization Phase: The clients initialize a transaction with the participating servers. The clients then initialize, potentially asynchronously, a number of sub-transactions. This phase is indicated by the `begin_tx` and `begin_sub_tx` messages in Figure 1 (the pink and green arrows at the top).

2) Operation Phase: The clients perform the operations for which they have initialized sub-transactions such as writing data, reading data, or performing a configuration change such as a resource deployment or cleanup. This is indicated by the `operation` message in Figure 1 (the blue arrows). The client subordinates directly perform the operations with the participating server resources. This phase, along with the Request Commit Phase will be repeated for each operation that is part of the overall transaction.

3) Request Commit Phase: This is a global coordination to determine if the sub-transaction is successful or should be aborted. How this is determined is based on the kind of operation performed. If it is a data output, then a count of processes and maybe a collection of checksums and/or total data sizes may be sufficient. If it is a system reconfiguration, some status indicator showing that a set of services are now available may be required. This is seen by the `req_commit_sub_tx` and `commit_sub_tx` messages (the green arrows in the middle of the figure).

4) Finalize Phase: Once all of the sub-transactions are complete, the overall transaction can be finalized to persist and expose the changes and operations performed as part of the transaction. The commit or abort for the transaction depends on whether or not all sub-transactions have completed successfully. This will include the ability to update any data storage or resource directory to reflect the completed transaction. This is seen in the `req_commit_tx` and `commit_tx` messages (the pink arrows at the bottom of the figure).

IV. Initial Protocol Implementation

The logical protocol is straightforward, but not scalable. The communication volume between the clients and the servers is a Cartesian product of the number of clients and servers making it grow very quickly as the scenario scales. In an effort to perform some initial measurements and determine where the real problems may be for scalability, some initial decisions are made for this first implementation that may or may not prove correct at scale. Ultimately, these decisions are being revisited to verify the scalability and performance as the implementation evolves. None of these initial decisions are necessarily inherent to the protocol itself and are largely a reflection of the development environment.

The first decision attempts to address the scalability of the coordination communication between the clients and the servers. This implementation aggregates this communication

through a coordinator process on each side. This dramatically reduces the message volume between clients and servers, but at a cost. The single coordinating process on each side is a new single point of failure that introduces additional failure modes that must be addressed. While the complication in failure modes causes additional implementation work, the current implementation's use of timeouts and voting for new coordinators on failures can adequately address these complications without undue problems. The single coordinator also only reduces the messaging load and does not fully solve the scalability challenge of coordination messaging between clients and servers. At scale, even a single, small message from each client to the coordinator is likely to cause both a count and aggregate size overload for a single process to handle. Several techniques, such as aggregation trees [20] and using a subset of processes for coordination, can address the messaging problem and are being investigated for the next iteration of the implementation.

The second decision is the choice of NSSI, further described in Section V-A, to communicate between the clients and servers and MPI to implement the messaging among the clients or servers. Directly related to this decision is the performance impact of using two different communication mechanisms to and attempting to detect messages. Indirectly related to this decision is the impact on programming models for a shared services-style application. Both of these are discussed in detail in Section V-B.

Others decisions give a stronger base for future work with D^{2}T. For example, the use of NSSI to communicate between clients and servers offers the advantage of separate process spaces for the clients and the servers. This isolates faults in the server area or in the client area from the other side. This affords clients continuing even if a server process has failed, improving the fault tolerance of the system. The use of timeouts to detect faults, while it may prove problematic at scale, is another example of a decision that is intended to survive the evolution of the implementation.

A. Global Protocol Decisions

For our initial implementation, a few simplifying assumptions are made. First, all client processes participate in all transactions and sub-transactions. The bookkeeping required to track a subset of processes was deemed optional because this implementation is intended to give insights into scaling and will focus on the highest message volume case. Second, all client processes are in sync in terms of transaction and sub-transaction IDs avoiding additional communication needed for the clients to agree upon a set of valid IDs. In practice, the clients will likely be able to predict these IDs rather than having to coordinate every time. For example, an output action is typically done by a fixed set of processes allowing each to keep a private counter for the current transaction ID. As a motivating example, scientific simulations typically work on a number of large arrays where each process is performing some computation on the local portion of the array values. At each output step, the processes write any number of variables, some of which are the local pieces of global arrays while others which are single value variables. Additionally, some metadata will be written, such as which portion of the global array this process is writing and how many elements are in this local portion of the global array. This information can later be used to index the data making it available for queries.

These simplifying assumptions, however, are not inherent to D^{2}T, but just for the initial implementation to generate initial measurements to reveal the scalability bottlenecks of distributed MxN transactions. As the system evolves, these assumptions are being relaxed to expand the variety of applications supported and to support more complex patterns. Ultimately, this protocol will be sufficiently general to be used for a wide variety of tasks, particularly complex data movement such as what is performed by adaptive mesh refinement codes as well as non-data movement activities that should be protected in an atomic operation.

The initial technique to detect a failed process in either the clients or servers is through a timeout. If at any time during this transaction process a timeout occurs, a subordinate aborts the sub-transaction or transaction locally and informs the coordinator. The coordinator detects this abort and disperses the message accordingly.

B. Implementation Details

As outlined above, D^{2}T consists of four phases for a transaction. Each of these is discussed in detail below in the context of how it exists in the current, initial implementation.

1) Initialization Phase: The initialization phase is represented by the `begin_tx` message in Figure 1. This consists of all of the clients participating in the transaction coordinating with the server processes that are participating. To accomplish this in the current implementation, the participating clients synchronize with the client coordinator, this is forwarded to the server coordinator. Next, the server coordinator notifies the participating server processes collecting `OK` messages. If all server processes respond, this is forwarded back to the client side coordinator.

If the initial coordination creating the master transaction is successful, the client coordinator can then initialize a sequence of sub-transactions with the server coordinator each representing an operation, such as writing a variable or allocating some system resources. The process is similar for sub-transaction initialization, but with the addition of an extra identifier field.

2) Operation Phase: The operation phase involves performing the sub-transactions and the operations they wrap, symbolized by the `operation` message in Figure 1. After receiving a success message for each sub-transaction initialization, the client coordinator informs its subordinates that it can start the operation(s) for each of these sub-transactions. The client subordinates can then start.

It should be noted here, that since every client processes is participating in a given sub-transaction and that it is unlikely for an application to want to proceed if only some sub-transactions can be instantiated and others cannot, it might be possible for the client coordinator to disperse a single success message to its coordinates. This is possible because of the assumptions mentioned above, however, in the future, it will

provide the application with the flexibility for some processes to create sub-transactions on the fly. A common use-case for this is the automatic mesh refinement (AMR) applications where some events are triggered on a sub-set of the processes causing those processes to have additional output.

3) Request Commit Phase: After each process completes the operation(s), the client coordinator asynchronously issues to the server coordinator a `req_commit_sub_tx` request for each sub-transaction. The server then validates that the sub-transactions were completed successfully with the participating servers. The server coordinator then responds to the client side saying it is ready to commit using the `commit_yes` message (or `commit_no` for failure). When the client coordinator receives a `commit_yes` message, it then broadcasts this message to the subordinates. The subordinates now mark their internal data structures that this sub-transaction is ready to commit and respond to the client coordinator with an `OK` message, identifying that the client subordinates can commit the sub-transaction. If the client coordinator then receives an `OK` message from each subordinate, the client coordinator sends the `commit_yes` message to the server coordinator. Another round of messages occurs on the server side to perform the actual commit operation for the sub-transactions and a final `OK` or error message is sent back to the client coordinator; this message is then dispersed to the subordinates. This completes the sub-transactions leaving the overall transaction ready for completion or abort.

4) Finalize Phase: After the request commit phase for the sub-transactions, the application can commit or abort the entire master transaction. If all sub-transactions have completed successfully, then it is natural to commit the transaction. However, if some sub-transactions did not complete, the application can make a decision as to whether or not to continue with a partial success or not, as it now knows which sub-transactions were problematic. Perhaps it could attempt to redo those operations or it could end up aborting the entire transaction all together. It may also be the case that having some servers fail that were scheduled to be decommissioned as part of the transaction safely can be ignored. The procedure for voting on a main-transaction is similar to the voting in the sub-transactions with a few rounds of messages and a final message between the client and server coordinators. A further wrinkle with this round of messages is the possibility that the servers involved in one of the sub-transactions successfully completed early sub-transactions, but failed later sub-transactions. This outer transaction loop affords an opportunity to manage the sub-transactions as a group so that decisions affecting the earlier sub-transactions can be managed as a part of the greater transaction. Once this phase completes, the operation(s) are both complete and correct and can be safely revealed to the rest of the system.

C. Applying to Motivating Examples

Given the described protocol and implementation, it is important to map these ideas back to the motivating examples so that we demonstrate how the ideas address these issues. First, we describe the role and function of D^2T in online workflows and code coupling. Then, we describe how dynamic system reconfiguration for fault tolerance or load balancing can be enhanced through using these techniques.

1) Online Workflows and Code Coupling: The initial work for data staging and code coupling is evolving towards online workflows and incorporating tight, frequent data exchanges between components. Towards that end, the functionality and much of the safety offered by centralized persistent storage must be addressed to help users trust moving online.

In the simple case of just moving data to a staging area for processing or retrieval by another process, the master transaction represents an entire data output event, such as a data dump for analysis, while the sub-transaction represents an individual variable from one or more participating processes.

At a detailed level, many other attributes must be considered. For example, since not all variables being output will be distributed across all processes in the D^2T, some additional features are required. To handle variables distributed on a proper subset of processes participating in the transaction, the setup for each sub-transaction specifies the number of clients that will participate for this sub-transaction. In a security suspect environment, the list of participating clients could be included to afford verifying a client is expected. Either of these approaches tells servers how many clients to expect rather than expecting all clients to participate. On the server side, this information can be used to determine if all expected clients have sent data or not. This also affords the opportunity for automatic mesh refinement codes with multiple levels that are distributed across a subset of the processes, but likely more than a single process.

Once the single data staging interface with transactions is complete, the next step towards online workflows requires handling a read and lock a data set, process it, and then write the processed data and delete the original. More concretely, for the CTH shock physics code in use at Sandia, if the raw data were written to a staging area using transactions, the next step would be to distill that data down to a list of fragments. To do this, the raw data needs to be read, marked as in process to prevent another parallel analysis process from accessing it, distilled down to a list of fragments, persisted back to a possibly different staging area, and if all of that was successful, unlock and delete the original raw data and release the list of fragments for further processing. This multi-stage process incorporating potentially multiple staging areas is key to being able to emulate the kind of functionality provided by using centralized, persistent data storage between workflow steps. In that case, the storage system offers some failure detection to help manage the evolution of data through the workflow, but it does not offer the way to hide/protect data to ensure improper or early access to incomplete data. In that way, using D^2Ts are superior to using a storage system for an offline workflow. The area where this is not true is persistence.

To handle persistence, it is possible to do one of several things easily and still achieve the more convenient and higher performance functionality offered by D^2T. For example, data could be stored to a node-local SSD or other storage class memory, it could be replicated to other nodes, or even written to centralized persistent storage. This last option is not ideal

for performance, but it is an option for data that may be critical to protect. None of these options have been implemented in the initial system. One of the next increments for this work is to explore these persistence techniques to offer both options and a variety of features that may vary depending on the platform.

2) Fault Tolerance and Load Balancing: System reconfiguration offers a very different example that ideally requires similar transaction functionality. Much like the cross-staging area transactions described above, system reconfiguration requires interaction with potentially more than a single set of 'server' resources.

For a system reconfiguration action, such as replacing one set of online resources with a new set of resources, the setup of the new resources would be one sub-transaction with the tear down being a second sub-transaction. In more detail, a transaction is initiated to represent the reconfiguration contacting all of the resources that are involved in some form with this transaction. Then, a sub-transaction is initiated to create a new set of resources. Once those resources have been initialized and responded that they are ready, signaling the end of that sub-transaction, the next step starts. The teardown of the older resources is initialized as the next sub-transaction. This requires a special case in that the teardown does not actually cause these resources to be reclaimed. Instead, it triggers setup of the shutdown process, even while these resources continue to be used actively. Once this shutdown process is initialized, finishing this sub-transaction, the metadata resources identifying which resources to use is updated to remove the soon to be decommissioned resources in a sub-transaction. Once all of these are complete, including emptying any dependent queues, the overall transaction can be completed. Once that is done, the physical teardown of the old resources can be safely triggered. This prevents clients from seeing the new resources prior to these resources being properly initialized or from using resources that should no longer be used. While a technique like read-copy update [21] conceptually can offer some support for similar operations, it is not intended to work outside of a single node and would only apply to the metadata operation. It is still necessary to manage the resource allocation and cleanup beyond the metadata operation. Performing those operations with a level of safety and in a structured way offers more simplicity and a holistic solution to the macro operation.

D. Failure Modes

From the above protocol description, it can be imagined that there are several failure scenarios that can occur at different points in the protocol. Our system is designed to detect these failure modes on both the client and server sides. An examination of some of the potential failure modes is listed below with a discussion of how each is either addressed in the current implementation or in the design being fleshed out over time in our experimental system.

For both the client and server sides, there are two possible sources of failures: the coordinator and the subordinates. One important feature is that if a failure is detected, the client side has the ability to retry its transactions or sub-transactions and even vote on a new coordinator should it be safe to continue in a reduced capacity. For example, if during the write phase of a data staging transaction, a staging server is determined to be down, the client processes can retry the sub-transaction by writing its data to a different server. Previous sub-transactions could be aborted and retried if any of them used the now-failed server as well, thus saving the overall transaction.

- **Coordinator Failure:** Should a coordinator fail, currently the transaction as a whole will have to be aborted and retried. Through the use of a timeout, either the clients or the servers will detect that the coordinator is no longer responding indicating a failure. When this timeout occurs, a vote is held to select a new coordinator. For simplicity, the lowest ranked process id or some other automatic selection criteria implicit in the system can be used. Once the new coordinator is selected, it will notify the other side to reestablish the connection. Should this communication fail due to a simultaneous failure on both sides, a broadcast from the new coordinator to the other side will establish the connection.
- **Client/Server Failure:** Far simpler than the coordinator failing, either a client or server process is easier to handle. The way a client or server process failure is detected is during one of the following actions: i) a broadcast or point-to-point message from the coordinator to the process fails, ii) the coordinator has a timeout waiting for a process to indicate that it has completed an operation, and iii) the servers do not receive the expected connection and tell the server coordinator an aggregate total that is less than expected. In these cases, the coordinator can abort the current sub-transaction and decide if it should re-attempt the operation or abort the transaction as a whole. It can also consider which process failed to determine the potential impact on other sub-transactions to manage the broader impact of the failure. If the failure is on the server side, a retry from the client is possible. If it is on the client side, the state of the component, such as the simulation, as a whole needs to be considered. The application may be in a state with the failure requiring a restart from a checkpoint or some other resilience restart.

E. Mapping to ACID Properties

Since the term 'transaction' brings to mind the ACID properties, it is important to show how the D^2T protocol achieves, as much as possible, these properties. Since transactions are most associated with databases, a data staging for an online workflow scenario is used to show to what extent the ACID properties are achieved.

Support for atomicity, consistency, and isolation is straightforward. The way these transactions prevent visibility of any operation prior to the final commit prevents any other user or transaction from seeing or using data in an in progress transaction prematurely. By preventing visibility of the data prior to the final commit, the data will be made visible as an atomic action. For durability, the situation is a bit more complex, but certainly possible. For example, storage class memories on every node can offer a level of security as long as a node does not completely disappear when a failure occurs.

For more security without this additional hardware, replication to other nodes can be performed. There is additional overhead associated with this additional data movement and storage, but it should generally still be significantly faster and cheaper than moving to centralized persistent storage. Each of these techniques offers different benefits and costs not limited to the level of catastrophic failures they can tolerate without any loss. Overall, these techniques are not intended to necessarily completely replace centralized persistent storage. Instead, they are intended to replace as much usage of that slow, expensive storage as possible for intermediate data with an online solution that will ultimately lead to a faster time to solution at less cost.

F. What about BASE properties?

The BASE [22] properties of Basically Available, Soft State, and Eventual Consistency offer a more scalable and flexible model for many applications. The D^2T techniques are intended for a different environment than BASE properties can support. For our example, the BASE property of eventual consistency is sufficient if partial or outdated answers are sufficient for results, such as for an Internet search engine. In our scenario, we are focused on supporting online workflows and other high performance and distributed computing operations. For these scenarios, eventual consistency is insufficient. The lack of a guarantee of when a system might be consistent limits the decreased time to solution an online workflow offers. The transaction overhead is outweighed by the increased throughput of the online workflow. Moreover, this throughput in the online workflow is only possible with guarantees about data completeness and correctness. The BASE properties are insufficient for maintaining this throughput.

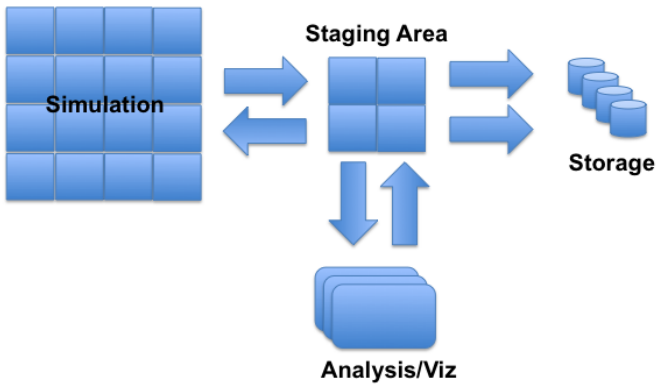

Fig. 2. Data Staging Overview

V. Performance Evaluation

To validate these ideas, two sets of experiments are performed. The first set tests how the overheads affect performance of a data staging scenario for an online workflow. Included with this is an examination of the impact of using two different messaging systems for the initial implementation. The second set are micro-benchmarks to measure the overhead of incorporating this overhead into a production system. This is representative of the overhead in a system reconfiguration scenario.

A. Experimental Setup

The experiments are performed on Sandia National Lab's RedSky unclassified machine. It is a Sun Blade center with Sun X6275 blades containing 2823 nodes running Intel Xeon 5570 processors (8 cores each) with 12 GB of RAM per node and QDR InfiniBand arranged in a 3-D toroidal mesh as the communication fabric. RedSky is a capacity cluster intended to run as many jobs as possible. Two communication APIs are used: Open MPI and Sandia's NSSI, an RPC package [23], [24] recently added to Trilinos as part of the Trios IO components that provides a simple API for an RPC mechanism that can manage RDMA data movements. It has native drivers for Portals, InfiniBand, and the new Cray Gemini networks.

The first set of experiments is designed to maintain a 128:1 ratio of clients to servers. This ratio is maintained for tests from 128 to 4096 clients where 32 server processes are employed. The experiments are performed 10 times for each setup and show the mean of the time required for each phase.

B. Online Workflows Results

To test the kind of overhead involved in online workflows, a payload size of 1 MiB is used. While larger payloads are typical for HPC applications, this gives a baseline that should be straightforward to extrapolate to larger payload sizes. An idea of how to scale the payload sizes for the NSSI protocol [25] on various platforms has been performed. The results of the experiments using 1 MiB payloads are shown in Figure 3.

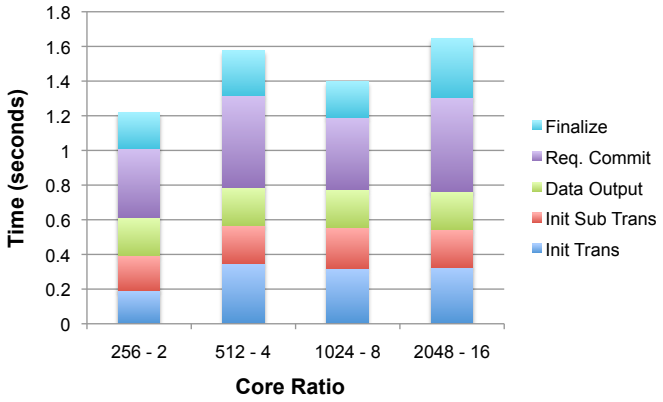

Fig. 3. Online Workflow Overhead, Client Perspective

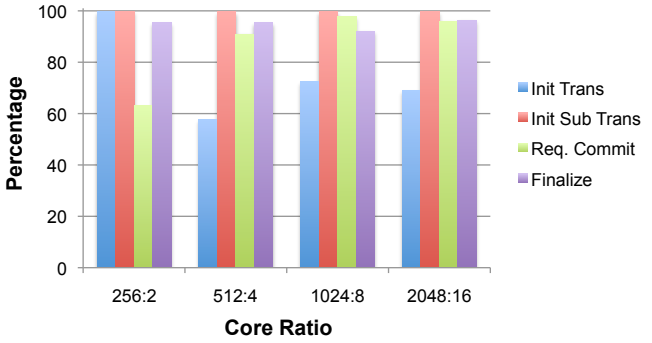

Fig. 4. MPI Overhead for Online Workflows

For this experiment a ratio of 128:1 clients to servers ratio is maintained as it scales.

By isolating the NSSI from the MPI messaging and viewing from the server side, the root cause of the overhead becomes apparent as shown in Figure 4. This figure shows the percentage of the overhead directly attributable to the MPI messaging as an artifact of the two messaging systems employed. First, the polling interval causes delays in receipt of messages introducing as much as 100 ms of delay for a round-trip message. This gives another reason why two messaging systems, beyond the current failure model of MPI, that it should not be used for this scenario–multiplexing between the two protocols to discover waiting messages introduces tremendous overhead compared to what would happen if a single mechanism were employed. The negligible overhead exhibited by MPI on the client side provides stronger evidence of this multiplexing issue. While the client uses both protocols, it only uses NSSI or MPI on an expected, rather than unexpected message basis eliminating the polling interval. Second, based on the limitations of which processes participate with each sub-transaction for the server, and the inability to use MPI_Probe to detect an MPI_Bcast message, individual MPI_Isend messages must be used instead. This, unfortunately, prevents any optimizations that may be implemented for efficient broadcasts from being used for this scenario, such as the specialized networks on the BlueGene platform. While the MPI 3 standard introduces non-blocking collective communication, this is not an adequate solution. The primary problem is that each process has no knowledge of what message may arrive next since multiple transactions may occur simultaneously. This would require a collection of pre-posted non-blocking collective calls each with associated buffers pre-allocated. While the number of different possible messages is not prohibitively large, the associated buffers of unknown size is too memory intensive for our limited memory environment. With a working MPI 3 implementation, it will certainly be possible to make large buffers and pre-post the collective calls and have it work, it is not ideal for this scenario. The advantage MPI_Probe for collective calls offers over the potential MPI 3 solution is the ability to detect the message size waiting in the unexpected message buffer and craft an accurately sized receive buffer. This affords having all potential messages share a single buffer rather than having to make individual buffers for each possible message.

C. Fault Tolerance and System Reconfiguration Results

For these experiments, the overhead of D^{2}T messaging is calculated for each of the phases. The operation phase depends on the underlying communication infrastructure and is independent of D^{2}T. There is no significant data movement as part of this example better representing the kind of overhead involved in this style of non-data movement transactions. The time the simulation spends at each phase of the protocol is measured. The results are shown in Figure 5.

At 128:1, D^{2}T spends very little time executing the protocol, as the server side does not have to poll or process any MPI messages. The jump between 128:1 and 256:2 is largely due to the introduction of subordinates on the server side. This introduces a polling delay of $(0, 100]$ milliseconds for a round trip MPI message based on the 50 ms poll interval. The slight variations in time periods, i.e., the time being slightly lower as the scale increases from 512 to 1024 client processes for example, is because of the server polling rate of the servers. As mentioned above, for a round trip message, anywhere from $(0, 100]$ milliseconds can be spent in between polling periods. As the experiment scales, each phase of the protocol increases at about the same rate as the other phases. This is because the number of round trip messages and message sizes are approximately the same, with only a few bytes difference.

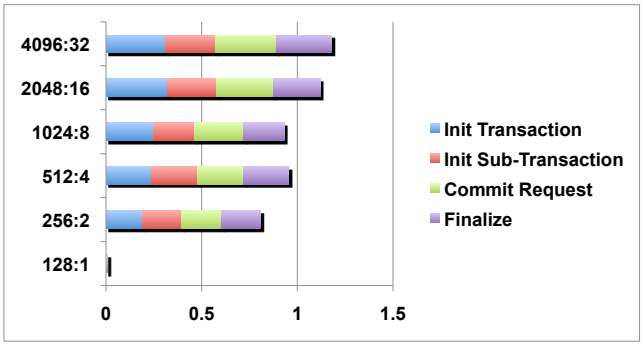

Fig. 5. System Reconfiguration Overhead

D. General Discussion

As the results show, D^{2}T does achieve good scalability; doubling the core count does not result in a doubling of the time the simulation spends in each phase of the protocol. This is partially due to using MPI to communicate between coordinator and subordinate, taking advantage of the efficiency in the implementation of the MPI library. Eliminating MPI to achieve fault tolerance and eliminate the delays will require replicating the messaging efficiency inherent to the mature MPI implementation.

Further improvements can be made to D^{2}T by adding different optimizations, such as batching sub-transaction initialization requests or piggybacking messages on top of each other as is done in Sinfonia [16]. For example, instead of creating sub-transactions synchronously, it would be beneficial to allow the simulation to create a large number of sub-transactions at once and sending these requests in one message to the server coordinator. Additional improvements could result from piggybacking messages, for example, piggybacking the request commit request of a sub-transaction with the operation sent in the operation phase.

To coordinate the different message queues, the current implementation must poll two queues. The first is for NSSI messages and the second is for MPI messages. However, there is some delay for how often the servers poll for MPI messages. For these experiments, the servers check for MPI messages approximately every 50 milliseconds. At higher intervals, the delay time begins to dominate and overshadow the protocol overheads as we scale in core count. Lowering this polling duration too much will steal time away from the server polling for NSSI messages. An initial set of quick tests suggests this as a good initial value. Considering this timeout delay is not intended to be a long-term solution no effort has been made

to exactly quantify what an ideal timeout delay would be for different core counts and configurations. Given that MPI is unlikely to be maintained for local messaging, it further discourages attempting to quantify the best interval.

VI. Conclusions and Future Work

This is very early results showing the potential of incorporating MxN doubly distributed transactions as a way to enable online scientific application workflows or dynamic system reconfiguration operations. We are extending this current implementation to be able to read completed transactions out of the staging area requiring at least a simple metadata system. Existing metadata solutions will both inform and hopefully supply a usable system for this purpose.

Durability approaches under investigation focus on node-local and compute area solutions. Avoiding the centralized storage system is a primary goal to ensure performance and scalability of this system.

The current reliance on timeouts for detecting failures is a convenience rather than a requirement. Other mechanisms that are less sensitive to jitter, such as ping messages, may be incorporated as the implementation progresses. These techniques will be selected based on the reliability and performance implications to the overall protocol.

There are several opportunities to optimize D^2T by piggybacking certain messages and providing an optional optimistic and potentially implied success model, thus reducing the overall volume of messages. Some examples of similar optimizations were found in Sinfonia, where they introduce the concept of mini-transactions. One such example found in this work is piggy-backing the transmission of the data along with the commit/abort request.

VII. 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] S. Ku, C. S. Chang, M. Adams, E. D. Azevedo, Y. Chen, P. Diamond, L. Greengard, T. S. Hahm, Z. Lin, S. Parker, H. Weitzner, P. Worley, and D. Zorin, "Core and edge full-f ITG turbulence with self-consistent neoclassical and mean flow dynamics using a real geometry particle code XGC1," in *Proceedings of the 22th International Conference on Plasma Physics and Controlled Nuclear Fusion Research*, no. IAEA-CN-165/TH/P8-40, Geneva, Switzerland, 2008.

[2] W. X. Wang, Z. Lin, W. M. Tang, W. W. Lee, S. Ethier, J. L. V. Lewandowski, G. Rewoldt, T. S. Hahm, and J. Manickam, "Gyro-Kinetic simulation of global turbulent transport properties in tokamak experiments," *Physics of Plasmas*, vol. 13, no. 9, p. 092505, 2006.

[3] NCAR and UCAR, "Community earth system model," http://www.cesm.ucar.edu/models/cesm1.0, 2012.

[4] 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.

[5] C. Docan, F. Zhang, M. Parashar, J. Cummings, N. Podhorszki, and S. Klasky, "Experiments with memory-to-memory coupling for end-to-end fusion simulation workflows," in *CCGRID*, 2010, pp. 293–301.

[6] N. Podhorszki, S. Klasky, Q. Liu, H. Abbasi, J. Lofstead, K. Schwan, M. Wolf, F. Zheng, C. Docan, M. Parashar, and J. Cummings, "Plasma fusion code coupling using scalable i/o services and scientific workflows," in *In Proceedings of The 4th Workshop on Workflows in Support of Large-Scale Science at Supercomputing 2009*, 2009.

[7] H. Abbasi, M. Wolf, G. Eisenhauer, S. Klasky, K. Schwan, and F. Zheng, "Datastager: scalable data staging services for petascale applications," in *HPDC*, D. Kranzlmüller, A. Bode, H.-G. Hegering, H. Casanova, and M. Gerndt, Eds. ACM, 2009, pp. 39–48.

[8] 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.

[9] C. Docan, M. Parashar, and S. Klasky, "DataSpaces: An interaction and coordination framework for coupled simulation workflows," *HPDC '10: Proceedings of the 18th international symposium on High performance distributed computing*, 2010.

[10] C. Isert and K. Schwan, "Acds: Adapting computational data streams for high performance," in *Proceedings of the International Parallel and Distributed Processing Symposium*. IEEE Computer Society Press, 2000, pp. 641–646.

[11] K. El Maghraoui, T. Desell, B. Szymanski, and C. Varela, "Dynamic malleability in iterative mpi applications," in *Cluster Computing and the Grid, 2007. CCGRID 2007. Seventh IEEE International Symposium on*, may 2007, pp. 591 –598.

[12] J.-P. Banâtre, M. Banâtre, and F. Ployette, "Construction of a distributed system supporting atomic transactions," in *Symposium on Reliability in Distributed Software and Database Systems*, 1983, pp. 95–99.

[13] 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.

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

[15] W. Allcock, J. Bresnahan, R. Kettimuthu, M. Link, C. Dumitrescu, I. Raicu, and I. Foster, "The globus striped gridftp framework and server," in *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, ser. SC '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 54–.

[16] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis, "Sinfonia: a new paradigm for building scalable distributed systems," *SIGOPS Oper. Syst. Rev.*, vol. 41, pp. 159–174, October 2007.

[17] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "Pnuts: Yahoo!'s hosted data serving platform," *PVLDB*, vol. 1, no. 2, pp. 1277–1288, 2008.

[18] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, pp. 35–40, April 2010.

[19] S. Das, D. Agrawal, and A. E. Abbadi, "G-store: a scalable data store for transactional multi key access in the cloud," in *SoCC*, 2010, pp. 163–174.

[20] D. S. Greenberg, R. Brightwell, L. A. Fisk, A. B. Maccabe, and R. Riesen, "A system software architecture for high-end computing," in *Proceedings ofSC97: High Performance Networking and Computing*. San Jose, California: ACM Press, Nov. 1997, pp. 1–15.

[21] P. McKenney and J. Slingwine, "Read-copy update: Using execution history to solve concurrency problems," October 1998.

[22] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier, "Cluster-based scalable network services," in *Proceedings of the SixteenthACM Symposium on Operating Systems Principles*, Oct. 1997, pp. 78–91.

[23] R. A. Oldfield, P. Widener, A. B. Maccabe, L. Ward, and T. Kordenbrock, "Efficient data-movement for lightweight I/O," in *Proceedings of the 2006 International Workshop on High Performance I/O Techniques and Deployment of Very Large Scale I/O Systems*, Barcelona, Spain, Sep. 2006.

[24] N. S. S. Interface, "https://software.sandia.gov/trac/nessie/."

[25] 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.