SANDIA REPORT

SAND2011-8739
Unlimited Release
Printed November 2011

Resilient Data Staging Through MxN Distributed Transactions

Jai Dayal, Gerald Lofstead, Karsten Schwan, Ron Oldfield

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

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.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2011-8739
Unlimited Release
Printed November 2011

Resilient Data Staging Through MxN Distributed Transactions

Jai Dayal
Georgia Institute of Technology
College of Computing
266 Ferst Drive
Atlanta, GA 30332-0765
jdayal3@gatch.edu

Gerald Lofstead
PO BOX 5800 MS 1319
Albuquerque, NM 87185-1319
gflofst@sandia.gov

Karsten Schwan
Georgia Institute of Technology
College of Computing
266 Ferst Drive
Atlanta, GA 30332-0765
karsten.schwan@gatech.edu

Ron Oldfield
PO BOX 5800 MS 1319
Albuquerque, NM 87185-1319
raoldfi@sandia.gov

Abstract

Scientific computing-driven discoveries are frequently driven from workflows that use persistent storage as a staging area for data between operations. With the bad and progressively worse bandwidth vs. data size issues as we continue towards exascale, eliminating persistent storage through techniques like data staging will both enable these workflows to continue online, but also enable more interactive workflows reducing the time to scientific discoveries. Data staging has shown to be an effective way for applications running on high-end computing platforms to offload expensive I/O operations and to manage the tremendous amounts of data they produce.

This data staging approach, however, lacks the ACID style guarantees traditional straight-to-disk methods provide. Distributed transactions are a proven way to add ACID properties to data movements, however distributed transactions follow 1xN data movement semantics, where our highly parallel HPC environments employ MxN data movement semantics. In this paper we present a novel protocol that extends distributed transaction terminology to include MxN semantics which allows our data staging areas to benefit from ACID properties. We show that with our protocol we can provide resilient data staging with a limited performance penalty over current data staging implementations.

Contents

List of Figures

Chapter 1

Introduction

Many current scientific computing applications generate tremendous amounts of data. In fact, some applications running on current generation petascale machines are already generating terabytes of data every few minutes [13, 19]. Frequency of this data is expected to increase even further from increased memory images in larger runs and to manage resilience requirements. The projected move from petascale to exascale [10] shows a 1000× increase in compute performance, a 100× increase in memory capacity, and only a 10× increase in I/O bandwidth. To realize the full potential of these exascale machines, the I/O performance problem must be addressed. Data staging techniques have demonstrated the ability to help alleviate this problem.

A looming barrier to adoption of these techniques is the movement of existing offline scientific workflows to use data staging areas as the intermediate storage location for data as it is processed by various workflow components. However, before these workflows can effectively be moved online using staging areas, failure must be addressed. One aspect that this paper covers is the need to 'know' that a data set has moved successfully, that it is complete, and that it is correct. Ideally, a level of durability is also required, particularly for the short term. In the long term, this will be less important as the relative cost of computation compared with data storage shifts strongly in favor of recomputation rather than storage. Key to enforcing these data guarantees is a protocol to manage the whole data movement such that these guarantees are enforced both writing and reading processes.

The benefits for using staging have been proven in several papers [16, 9, 18]. Our group has taken the data staging approach using asynchronous I/O and hosting operations on data prior to writing to persistent storage [1, 20]. Some of the operations we have demonstrated a decrease in total compute time even when including the additional cost of the staging nodes performing indexing, filtering, and preparing/formatting the data to decrease subsequent read-times for analysis or visualization toolkits [20, 8]. This "in-flight" analysis gives scientists earlier access to the data aiding and accelerating data validation and the scientific discovery process.

One limitation to this approach, however, is that with such a large number of compute resources, faults are expected regularly rather than as a rare exception. Since we are writing to another processes memory instead of directly to permanent storage, any data stored in a processes memory can be lost during a fault or node crash. Depending on the constraints, some simulations and analysis tools require complete data sets for downstream processing.

For example, a complete data set is required to completely visualize an output set without loss of information or fidelity. The portion of resilient staging this paper addresses is providing transactional support for the data movements between the simulation, analysis code, and staging area. To do this, we augment current transaction terminology, which follows 1xN semantics, to work with MxN scenarios, i.e., M simulation processes writing to N staging server processes.

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. This initial approach adds a few extra rounds of messages during a scientific application's output phase 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 as data is moved to, and stored in, the staging area. Our current implementation uses two popular communication APIs to transmit the data and metadata both locally and between the simulation and the staging process spaces. Communication between simulation and staging area is done via the NSSI RPC package [17, 11]. NSSI was recently added to Trilinos as part of the Trios I/O capability area. It provides a simple API for an RPC mechanism that can manage RDMA data movements. It has native drivers for Portals, InfiniBand, Cray LUC, and the new Cray Gemini networks. By using a separate process space for staging, faults in the staging area or in the compute area are isolated from each other avoiding the loss of one due to the failure of the other. Communication within the simulation and within the staging area is performed using traditional MPI messaging. Neither of these choices are a requirement for the protocol to function properly.

Multiple advantages can be gained from adding resilience to data staging. By encapsulating our data movements into transactional units, we can hide from any readers, such as an analysis or visualization code, perhaps another running simulation, incomplete or incorrect data sets. We can provide an application with knowledge as to whether or not its data has been successfully committed to staging (as in all data has been written). We can more quickly identify failures so the application can better decide how to proceed, such as deciding to change the output to writing to persistent or different storage to avoid loss of this data output. The last decision can be handled in the IO API rather than requiring any intervention from the scientific application programmer. A system like ADIOS [15] that affords incorporating these custom protocols makes adding this protocol transparent.

Through these techniques, we enable data staging to move from the realm of solely being used as a way to hide I/O costs or to perform 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.

The remainder of this paper is organized as follows. Chapter 2 presents a short overview of the related work in the field. We introduce the concept of resilient data staging and MxN transactions in chapter 3. We next present our design and implementation in chapter 4, as well as a discussion on the different failure modes and how we detect these failures in our system. Chapter 5 presents our results, and chapter 6 presents our conclusion.

Chapter 2

Related Work

Much research has been conducted on providing resilient distributed data stores and transport mechanisms. However, while such work has provided novel benefits for their intended platforms, they lack several key features needed for our data staging use cases.

GridFTP [3] extends traditional FTP to provide reliable high-performance data movement in a grid computing environment. GridFTP provides support for collective data transfers via parallel striped data-transfers, where files distributed over several storage devices is transferred over some number of channels to a set of receivers. GridFTP also provides a way to restart transmissions that have been interrupted, so that the entire data transfer does not have to start over from the beginning.

This work differs from ours on several fronts. First, our protocol is designed to operate during the on-going simulation in a time-critical environment, where as GridFTP is designed to transfer data before and after simulation runs. Our work is also designed to work at extreme scales, with potentially millions of cores on one side communicating with thousands on the other. It's unseen if GridFTP can perform at these scales. Additionally, to our knowledge, it's not completely clear as to what level of safety semantics GridFTP provides, for example, data users can see files that have incomplete data from interrupted transmissions. Our intent is to shield data consumers from such erroneous data.

We have also surveyed a range of work for resilient distributed systems more geared towards the enterprise community, such as Sinfonia [2], PNUTS [5], Cassandra [14], and G-Store [7]. While these systems 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, these systems make use of disk storage devices for logging, which helps to provide durability and persistence. Data staging is intended to shield the simulation from disk overheads, so using log-files in such a manner may reintroduce these overheads. Third, work like PNUTS take advantage of eventual consistency models which will not work for our HPC environment, as allowing analysis codes or visualization tools to operate on stale data is useless and expensive. The potentially infinite delays for the eventual consistency to occur can also inject unacceptably long delays in processing both from an interaction perspective, but also from a data storage perspective. If the consistency is delayed too long, the amount of storage must increase to deal with the incomplete data set whie the next interation may begin movement to the staging area.

In summary, the key differentiators between our work and the other research, including works such as G-Store and Cassandra is that we require MxN semantics and immediate consistency for distributed transactions. Staging areas are intended to shield the simulations from storage system overheads, so use of log-files is troublesome, and we intend to operate at extreme scales, with potentially millions of cores; it's not clear of the previous research can scale to these levels.

Chapter 3

Resilient Data Staging

Figure 3.1 presents a conceptual model of a data staging area. Our view is a departure from existing views of staging areas in that we see storage as a last resort and instead are focused on building support for complete, in compute area workflows consisting of a core simulation that has data processed by a collection of analysis codes through a staging area. What we have is some group of processes reading and writing to a data store, which is composed of some other group of a number of processes, known as an MxN data redistribution [6, 12]. Additionally, there might be some visualization or analysis engine reading and writing to the data store. Figure 3.1 also shows a storage subsystem that would be used to permanently store the data for later access and use. The staging area can be viewed as a type of intermediary between the simulation, various analysis and visualization routines, and the shared storage device. Although the diagram only shows a single staging area, nothing in this design precludes using multiple staging areas to move data through the online workflow while minimizing interference effects from network contention.

Using the staging area provides us with several benefits. With the traditional straight-to-disk approach, the disk subsystems are shared among at least other processes on the same machine if not also processes on other machines that share the same storage array. This introduces contention points that further degrade I/O performance. With staging, we have direct control over our staging resources so we can employ our own resource, fault tolerance, and data management strategies as needed. Additionally, because the staging servers are compute nodes, we can uses these compute resources to perform useful operations while the data is on its way to disk [20]. Finally, we can choose how many different processes use the same staging area managing the contention for both bandwidth and memory resources. This final piece is what will afford sufficient memory to support the entire offline workflow process while maintaining I/O bandwidth. Ultimately, we would like to see this work as a piece of the argument for incorporating large memory capacity staging nodes into future HPC platforms.

This new approach introduces some drawbacks in regards to fault tolerance and resilience; it lacks the ACID guarantees that can be found in modern storage subsystems. With these ACID guarantees, we can provide applications and analysis codes with some guarantee that the data has been moved completely and correctly and that once the operations are completed, they are not lost. While the presented protocol supports the atomic, consistent, and isolated properties, supporting durability requires additional functionality such as replication

and node local persistent storage. Distributed transactions are a proven method for providing ACID properties, so we leverage them here and extend upon them, as current distributed transactions operate with 1xN semantics, to operate with our MxN data movements.

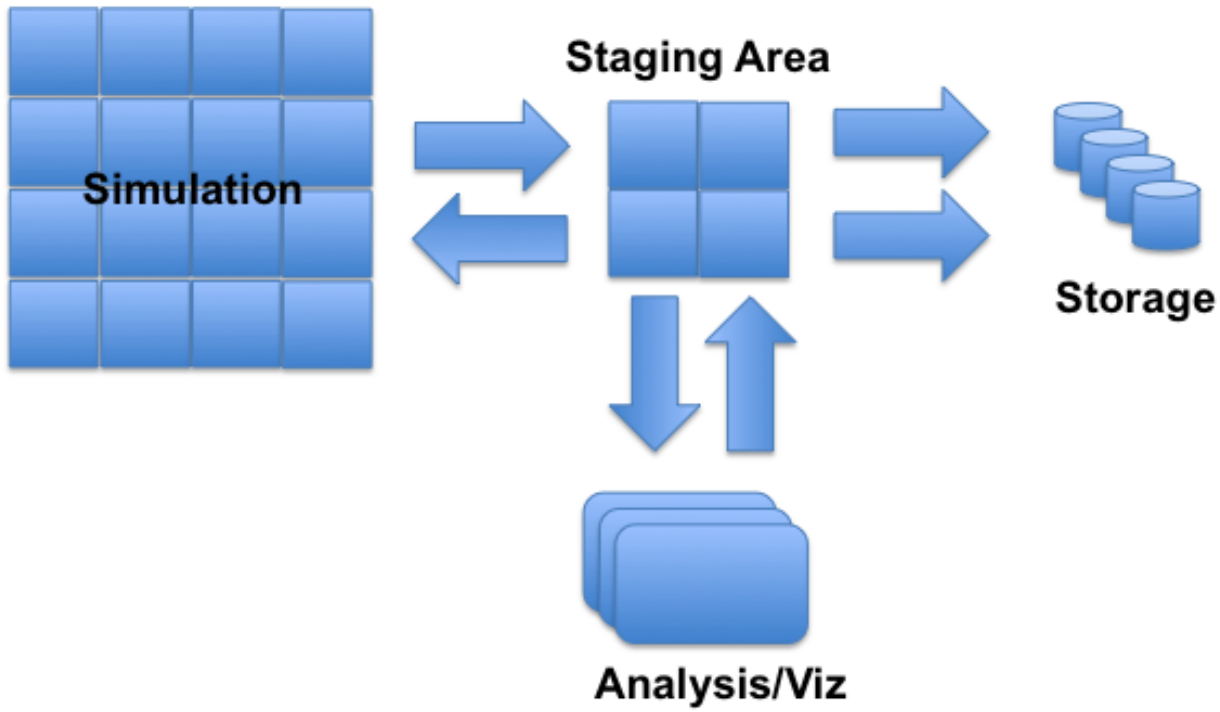

Figure 3.1. Data Staging Overview

Chapter 4

Design and Implementation

Assumptions

For simplicity in our initial implementation, we made a few assumptions. All client (compute) processes participate in all transactions and sub-transactions. All client processes are in sync in terms of transaction and sub-transaction IDs; we avoid additional communication needed for the clients to agree upon a set of valid IDs if this knowledge is not known. We are considering applications that are typically working on a number of large arrays where each processes 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 assumptions, however, are not inherent to our idea, but just for our initial implementation so we can get some benchmarks and some idea as to the scalability of distributed MxN transactions. As we are extending the system, these assumptions are being relaxed so we can operate with a wider variety of applications and support more complex I/O patterns. Ultimately, this protocol will be sufficiently isolated from the IO stream to be used for dynamic system reconfiguration tasks and other non-data movement activities that should be protected using ACID properties.

Protocol

The communication for our MxN transaction protocol is implemented with, but not tied to, two communication mechanisms: NSSI [11] and MPI. NSSI is an RPC framework built as part of the Lightweight File Systems Project (LWFS) [17]. NSSI supports high performance communication technologies such as InfiniBand, Portals, and GNI. Figure 4.1 is an overview of our protocol. For lines with no message description, it's the same description as the line above it.

Our goal here is to shield the simulations and analysis codes from failures with other components of the system. It is possible to implement our MxN transaction protocol strictly

with MPI, but this way has several drawbacks. First, if we make the staging area and the scientific application one MPI application, any process crashes in the staging area will bring down the simulation and visa-versa, violating the notion of protecting one component from the failures of another. Second, we could use MPI intercommunicators, however, these have performance implications as current MPI implementations only allow blocking point-to-point MPI calls.

For our protocol, we have two distinct sides, the compute clients (the simulation) and the staging application. In an initial effort to manage the MxN cartesian product sets of messages, the initial protocol implementation replaces this logical communication with a transaction coordinator for each side reducing the complexity to a 1-to-1 communication. This optimization affords an initial level of scalability, but has not been proven to scale to exascale-sized problems. Each transaction coordinator acts as a liaison to the other side, for example, only the client coordinator initializes a transaction with the staging coordinator. The coordinators are also responsible for making decisions for their respective sides (such as committing or aborting a transaction) and dispersing these decisions to the subordinates on their side.

We use NSSI to communicate between the compute processes and the staging processes. This includes communication between coordinators as well as when the compute clients write or read data from the staging servers. MPI is used when coordinators need to communicate to their respective subordinates, such as gathering information from the subordinates, or dispersing decisions to the subordinates. An example of such a decision on the server side, would be responding to a request_to_commit from the client coordinator, as we explain below.

To support transactions for the multiple variables that comprise a single data movement event in the HPC environment, we take a nested-transactions approach. A given transaction consists of any number of sub-transactions that represent the read/write of one variable or array piece in an output step. A sub-transaction consists of any number of read or write operations, and based on our assumptions above, for N client servers, the staging area as a whole will expect some multiple of N operations. For any given output step, there may be hundreds of sub-transactions. Our protocol consists of four phases for a transaction.

1. Initialization Phase - The clients initialize a transaction with the staging coordinator. The clients then initialize, potentially asynchronously, a number of sub-transactions with the staging coordinator. This phase is indicated by the begin_tx and begin_sub_tx messages in figure 4.1.

2. I/O Phase - The clients read and write the data for which they have initialized sub-transactions. This is indicated by the write_data message in figure 4.1. The client subordinates read/write directly from the staging subordinates.

3. Voting (Validation) Phase - Asks whether or not the sub-transactions and outer transaction can be committed or aborted. This is determined by a message count: if the staging subordinates have received, in total, the expected number of reads or writes

correctly, the transaction can be committed. If not, the vote on the staging side is to abort. This is scene by the request_vote_sub_tx message.

4. Finalize Phase - Commits or aborts the transaction, depending on whether or not all sub-transactions have completed successfully, updating any metadata store and data storage to reflect the completed transaction.

Initialization Phase

The begin_tx message represents the initialization phase as shown in figure 4.1. The client coordinator sends this message to the server coordinator that then broadcasts the message to the server subordinates. If not all OK messages are returned from the subordinates to the server coordinator, for an allotted time window, the server coordinator returns to the client coordinator an error message. The client coordinator forwards the server coordinators response to the client subordinates.

If the first step is successful, the client coordinator can then initialize a sequence of sub-transactions with the staging coordinator. The process is the same for sub-transaction initialization, with the addition of an extra identifier field. variable, it is possible for hundreds of variables to be written per client process per timestep, thus, there may be hundreds of sub-transactions. Unlike the previous step, the sub-transactions can be initialized at the client side in a non-blocking manner; the client initializes a batch of sub-transactions at once, and waits for the responses. subordinates in similar fashion to the first step.

I/O Phase

The next phase involves moving data, symbolized with the write_data message in Figure 4.1. After receiving a success message for each sub-transaction, the client coordinator informs its subordinates that it can start reading/writing data for these sub-transactions. The client subordinates then start asynchronously writing or reading their data.

It should be noted here, that since every client processes is participating in a given sub-transaction, and that it's unlikely for an application to want to proceed if only some sub-transactions can be instantiated and others can't, it might be possible for the client coordinator to disperse a single success message to its coordinates. This is possible because of the assumptions we mentioned above, however, in the future, we want to 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) scenarios where some events are triggered on a sub-set of the processes causing those processes to have additional output.

Voting (Validation) Phase

After each process has written its data, the client coordinator will asynchronously issue to the staging coordinator a voting request for each sub-transaction. The 'vote' consists of the staging coordinator asking the subordinates how many writes for a given sub-transaction it has received. If the total number of writes matches the number of expected writes, then the staging coordinator returns a COMMIT message to the client coordinator. If not, then an ABORT message is returned instead. This gives us some level of atomicity; the data movement for a sub-transaction is all or nothing.

Upon receiving a COMMIT message, the client coordinator then broadcasts this message to the subordinates. Upon receipt of this message, the subordinates will respond with an OK message and prepare to commit the transaction. If an OK message is received from each subordinate within the allotted time window, the client coordinator sends the commit message to the staging coordinator. Another round of messages occurs on the staging side, and a final OK or error message is sent back to the client coordinator; this message is then dispersed to the subordinates.

Finalize Phase

After the voting phase for the sub-transactions, the application can begin to commit or abort the outer transaction. If all sub-transactions 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 partial data or not, as it now knows which sub-transactions, or variables, were problematic. Perhaps it could attempt to re-write those variables or abort the entire transaction all together. The procedure for voting on a main-transaction is similar to the voting in the previous steps with a few rounds of messages and a final message between the client and staging coordinator.

It is important to note that if at any time during this process a timeout occurs, a subordinate aborts the transaction locally and informs the coordinator. The coordinator detects this abort and disperses the message accordingly.

Failure Modes

From the above protocol description, we can see that there are several failure scenarios that can occur at different points in the protocol. Our system is designed to detect these failure modes at the client side and staging server side. For each of the phases listed in section 4, failures can occur at both the data readers and writers and the data staging area. 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.

Failures can happen at the coordinators or at the subordinates, and we can detect these via timeouts and message counts. For example, timeouts are typically used by subordinates to determine if the coordinator has failed. For example, if a transaction has not had any state changes for a period of time, the subordinate will set the flag for its local object representing the appropriate transaction or sub-transaction as aborted. For a coordinator, if the coordinator does not receive the correct number of responses from its subordinates within a period of time, it considers this to be a subordinate failure, and will inform the remaining subordinates, and opposite coordinator, accordingly. Currently, we are working on adding some durability to ensure that the data persists even if a staging process dies after the transaction or sub-transaction has completed and been committed. We are also working on ways to allow clients to re-try failed sub-transactions by writing data to different staging servers, in case of a staging server failure.

Initialization Phase Failures

The initialization process for both transactions and sub-transactions follow the same steps, so the failures and detection apply to both. At each side (writers/readers and staging), there are two possible sources of failures: the coordinator or the subordinates. One important feature here is that as we detect failures, 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, a staging server is down, the client processes can re-try the sub-transaction by writing its data to a different staging server.

- **Client Coordinator Fails:** Subordinates on the client side detect this via a timeout; if the transaction is in limbo, or the same state, for too long, it is aborted. The staging servers have timeouts as well. If the staging coordinator cannot send the response back after a period of time, the staging coordinator will abort the transaction.
- **Client Subordinate Fails:** Client coordinator detects that it does not receive the correct amount of responses within the time window, so it aborts and tells remaining subordinates to abort the started transaction. Client coordinator also sends a message to the staging coordinator to abort. The staging servers do not know of such failures explicitly as the staging coordinator simply receives the abort from the client coordinator.
- **Staging Coordinator Fails:** The client coordinator knows this occurs when it stops receiving returns from its RPC calls. When this is detected, the client coordinator will tell its subordinates to abort. Staging subordinates have timeouts too, so they will cancel transactions in limbo for too long. Since the staging subordinates will no longer be receiving messages from the staging coordinator, the transaction will not be changing states.
- **Staging Subordinate Fails:** The staging coordinator does not receive the correct number of responses from its subordinates. When this occurs, the staging coordinator

sends an error to client coordinator as an RPC return value and tells the remaining staging subordinates to abort.

Writing Phase Failures

These are failures that can happen during the writing phase, i.e., when the clients are reading and writing data to and from the staging servers. If some read and write operations do not go through, then the number of successful operations will be less than the number expected. These errors can be detected as follows.

- **Client Side:** The client subordinates or coordinator does not get responses back after sending data to server. The data is sent via NSSI RPC call, and if no response is received after a period of time, this read/write operation is marked as unsuccessful. During the voting phase, the client coordinator asks each subordinate how many writes were successful, if the number of successes matches the number expected, the sub-transaction is marked successful, which means it can be later committed. It is true that the data might be transferred, but done so incorrectly resulting in junk data. The data can be validated by using MD5 hashes or checksums.
- **Staging Side** If a staging subordinate goes down after the messages are sent, we will detect this during the voting phase when staging coordinator asks each subordinate how many writes for variable it received. If some subordinates do not respond, the number of successful writes will not equal the number of expected writes.

Voting and Finalize Phases Failures

From the above discussions, we can see how the system uses a combination of timeouts and message counts to determine if a failure has occurred or not. For the remaining two phases, voting and finalization, the same methods listed above are applied here.

Alternatives to Timeouts for Fault Detection

While timeouts are the current mechanism used for detecting some failures (e.g., the loss of a process), other mechanisms can certainly work and can potentially dramatically reduce the number of messages required to enforce the guarantees. One initial mechanism under consideration is to rely on the reliability of the underlying parallelism mechanism, such as MPI, to detect process failures and ultimately abort the transaction. This sort of mehcanism is implementation dependent, but could potentially dramatically reduce the coordination messaging requirements compared with the current, more general implementation.

Other mechanisms that reduce either the message count and/or provide a way to passively detect a failure could dramatically improve the scalability of this technique. Approaches

that are also more general than relying on the underlying transport for parts of the failure detection are currently under investigation.

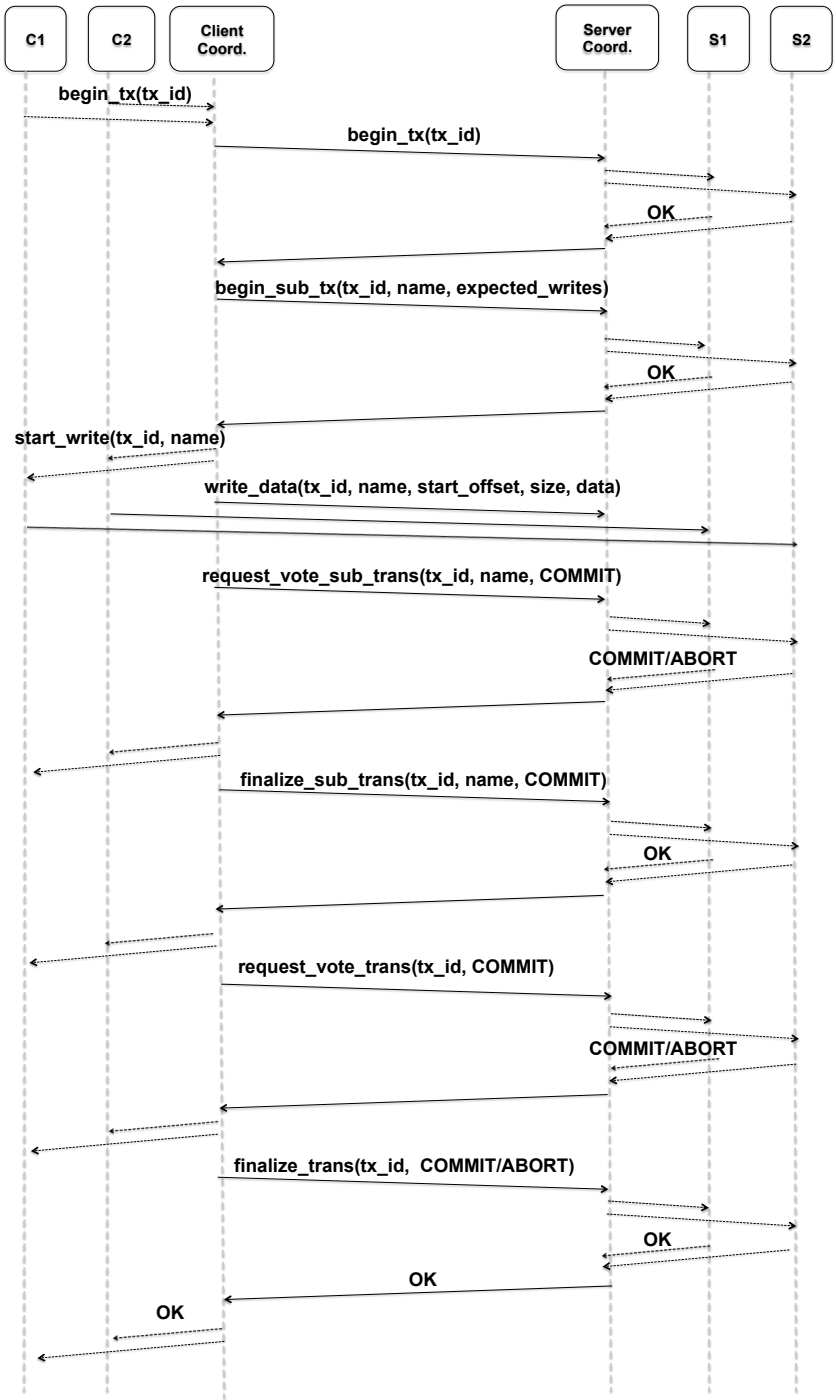

Figure 4.1. MxN Distributed Transaction Protocol

Chapter 5

Performance Evaluation

Experimental Setup

The experiments are performed on Sandia National Lab's RedSky machine, a Sun Blade center containing 4000 nodes, running Intel Xeon 5500 Series processors (8 cores each), with InfiniBand as the communication fabric. Two common communication APIs, OpenMPI and Sandia's NSSI, are employed as mentioned above. OpenMPI is used for communication between a coordinator and its subordinates. To communicate across application barriers, i.e., between MPI applications, Sandia's NSSI library is used. This shields the simulation from staging server failures and visa-versa. This also avoids the performance overheads of using MPI inter-communicators, which only allow for point-to-point, blocking communication

The staging servers poll a set of message queues: one set for NSSI messages and another set for MPI messages. This implementation introduces some delay when detecting the arrival of an MPI message. For these experiments, the staging servers check for MPI messages approximately every 50 milliseconds. At higher intervals, the delay time begins to dominate and overshadow the protocol overheads as the core count scales. Lowering this polling duration too much will steal time away from the staging server polling for NSSI messages. The selected 50 ms works as a balanced, realistic delay that must be revisited as the system scales.

Results

For these experiments, the overhead of the protocol is tested at each of the phases. The reading/writing phase depends on the underlying communication infrastructure and is independent of the protocol. The measurements show the time the simulation spends at each phase of the protocol, and measure this at different core counts, all using a ratio of 128 simulation processes to 1 staging process. As shown in figure 5.1, we scale from 128:1 to 4096:32.

At 128:1, the protocol 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 fact that now there are subordinates on the staging side and the polling delay can be any where between (0,100] milliseconds for a roundtrip MPI message. The

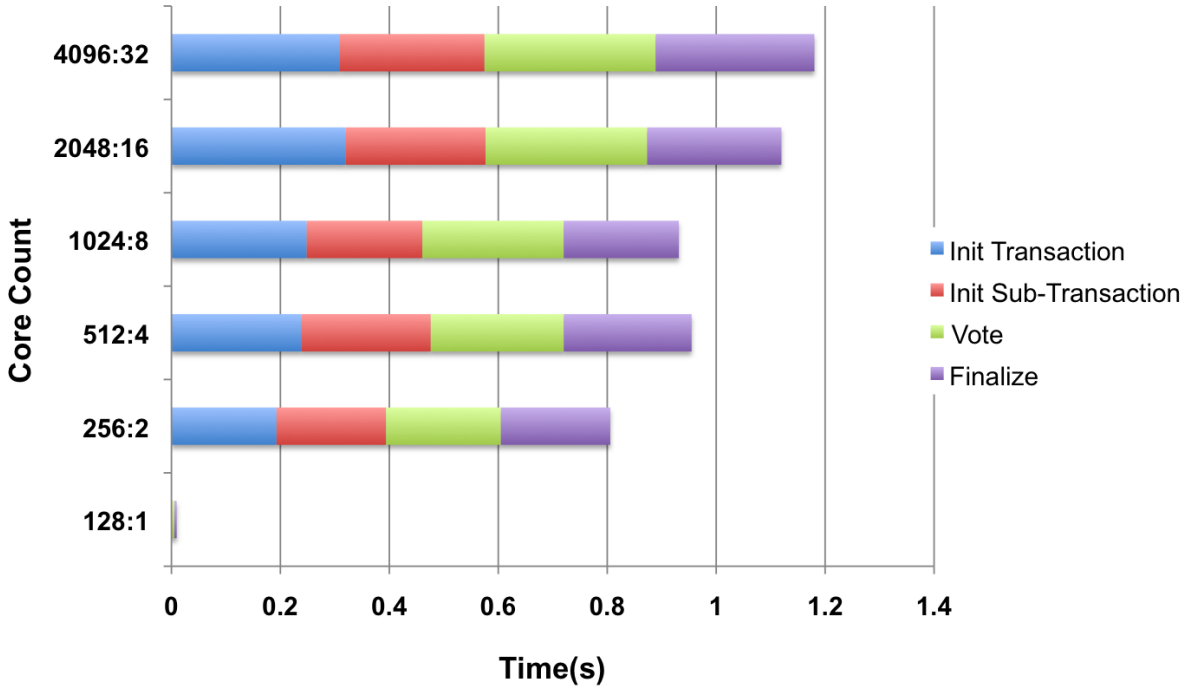

Figure 5.1. Protocol Overhead

slight variations in time periods, i.e., the time being slightly lower as it moves up in scale, between 512 and 1024 client processes for example, is because of the server polling rate of the staging servers. As mentioned above, for a round trip message, anywhere from (0,100] milliseconds can be spent in between polling periods. As this scales up from there each phase of the protocol increases at about the same rate as the other phases. This is because the number of roundtrip messages and message sizes are pretty much the same, with only a few bytes difference. In these experiments, the time required to initialize a sub-transaction is an average of 10 sub-transactions per transaction where each sub-transaction was created in a blocking manner. We are currently implementing a feature so that a simulation can instantiate a number of sub-transactions asynchronously.

As the results show, the protocol 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 largely due to MPI being used to communicate between coordinator and subordinate thus exploiting all of the advantages the scalability improvements implemented in the MPI library.

Further improvements can be made to the protocol by adding different optimizations such as batching sub-transaction initialization requests, or piggybacking messages on top of each other as in Sinfonia [2], a task currently underway. 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 staging coordinator. Additional improvements could result from piggybacking messages. For example, it is possible to piggyback the voting request of a sub-transaction with the last chunk of data sent in the write phase.

In summary, the implementation of the protocol, in its most straight-forward fashion with no attempt at incorporating optimizations, shows that the protocol does not add large overheads to the output phase of a simulation and that it can achieve good scalability using standard communication libraries like MPI.

Chapter 6

Conclusions and Future Work

As stated above, this is very early results showing the potential of incorporating MxN distributed transactions as a way to enable online scientific application workflows. We are extending this current implementation in several ways. First, we must be able to read completed transactions out of the staging area. To do this, we are working on a metadata service that indexes which transactions are committed and ready for reading and where these data pieces are located within the staging area. Existing efforts, like SciDB [4], in-memory databases, and HPC-related data storage formats metadata will guide the annotation, indexing, and query capabilities provided.

Our current implementation provides some level of atomicity; all pieces of data are written in full or the transaction is aborted, but we are moving towards full ACID compliance. One feature we are working on is adding some redundancy, such as data replication or parity storage similar to RAID systems, to provide durability so that if a server is lost, the data can be recovered. This can also be accomplished in the future as technologies involving non-volatile memory progress and are incorporated into future platforms. Traditionally, this is done with write-ahead log files on disk. However, our staging model is designed to avoid the overheads involved with writing to disk.

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 also several opportunities for us to optimize our protocol by piggybacking certain messages and providing an optional optimistic and potentially implied success model thus reducing the overall volume of messages. Some examples of similiar optimizations were found in Sinfonia [2], 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. Our work will make use of such ideas.

An important component that must be included as well to extend the viability of this work from the generally secure HPC environment to a distributed environment is the inclusion of data validation schemes, such as an MD5 hash, for each block written to the staging area. Care will be taken to help ensure not only random or failure induced changes are caught, but also some attention to malicious attacks on the data movement can also be addressed.

Additional data management optimizations, such as in staging area reorganization to reduce the number of data blocks, braking data into more manageable pieces, compression, statistical sampling, and other techniques as demonstrated in PreDatA [20] will also be incorporated to improve the performance and scalability of staging areas for online scientific workflows.

From the preliminary results shown in section 5, we can provide a level of resilience for data movements to and from staging areas by augmenting traditional distributed transactions to contain MxN semantics. To do this, we take a nested transaction approach, where a given transaction consists of any number of sub-transactions. Our results show that the messaging overhead our protocol induces is small enough so that we still retain performance gains over traditional directly to disk methods.

References

[1] Hasan Abbasi, Matthew Wolf, Greg Eisenhauer, Scott Klasky, Karsten Schwan, and Fang Zheng. Datastager: scalable data staging services for petascale applications. *Cluster Computing*, 13:277–290, 2010. 10.1007/s10586-010-0135-6.

[2] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. *SIGOPS Oper. Syst. Rev.*, 41:159–174, October 2007.

[3] William Allcock, John Bresnahan, Rajkumar Kettimuthu, Michael Link, Catalin Dumitrescu, Ioan Raicu, and Ian Foster. The globus striped gridftp framework and server. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, SC '05, pages 54–, Washington, DC, USA, 2005. IEEE Computer Society.

[4] Paul G. Brown. Overview of scidb: large scale array storage, processing and analysis. In Ahmed K. Elmagarmid and Divyakant Agrawal, editors, *SIGMOD Conference*, pages 963–968. ACM, 2010.

[5] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *PVLDB*, 1(2):1277–1288, 2008.

[6] Kostadin Damevski and Steven G. Parker. M x n data redistribution through parallel remote method invocation. *IJHPCA*, 19(4):389–398, 2005.

[7] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. G-store: a scalable data store for transactional multi key access in the cloud. In *SoCC*, pages 163–174, 2010.

[8] Ciprian Docan, Manish Parashar, and Scott Klasky. Dataspaces: an interaction and coordination framework for coupled simulation workflows. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 25–36, New York, NY, USA, 2010. ACM.

[9] Ciprian Docan, Fan Zhang, Manish Parashar, Julian Cummings, Norbert Podhorszki, and Scott Klasky. Experiments with memory-to-memory coupling for end-to-end fusion simulation workflows. In *CCGRID*, pages 293–301, 2010.

[10] Alan Gara. Energy efficiency challenges for exascale computing. In *Power Efficiency and the Path to Exascale Computing Workshop at SC 08*, 2008. http://www.lbl.gov/CS/html/SC08ExascalePowerWorkshop/gara.pdf.

[11] Network Scalable Service Interface. https://software.sandia.gov/trac/nessie/.

[12] Katarzyna Keahey, Patricia K. Fasel, and Susan M. Mniszewski. Paws: Collective interactions and data transfers. In *HPDC*, pages 47–54, 2001.

[13] S. Klasky, S. Ethier, Z. Lin, K. Martins, D. McCune, and R. Samtaney. Grid-Based parallel data streaming implemented for the gyrokinetic toroidal code. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 24, Washington, DC, USA, 2003. IEEE Computer Society.

[14] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44:35–40, April 2010.

[15] J. Lofstead, Fang Zheng, S. Klasky, and K. Schwan. Adaptable, metadata rich io methods for portable high performance io. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1 –10, may 2009.

[16] Jay F. Lofstead, Fang Zheng, Scott Klasky, and Karsten Schwan. Adaptable, metadata rich io methods for portable high performance io. In *IPDPS*, pages 1–10, 2009.

[17] Ron A. Oldfield, Patrick Widener, Arthur B. Maccabe, Lee Ward, and Todd Kordenbrock. Efficient data-movement for lightweight I/O. In *Cluster*, Barcelona, Spain, September 2006.

[18] Norbert Podhorszki, Scott Klasky, Qing Liu, Ciprian Docan, Manish Parashar, Hasan Abbasi, Jay F. Lofstead, Karsten Schwan, Matthew Wolf, Fang Zheng, and Julian Cummings. Plasma fusion code coupling using scalable i/o services and scientific workflows. In *SC-WORKS*, 2009.

[19] 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*, 13(9):092505, 2006.

[20] Fang Zheng, Hasan Abbasi, Ciprian Docan, Jay Lofstead, Scott Klasky, Qing Liu, Manish Parashar, Norbert Podhorszki, Karsten Schwan, and Matthew Wolf. Predata-preparatory data analytics on peta-scale machines.

DISTRIBUTION:

	MS	,
		,
1	MS 0899	RIM-Reports Management, 9532 (electronic copy)
1	MS 0359	D. Chavez, LDRD Office, 1911

Sandia National Laboratories