SANDIA REPORT

SAND2021-1451
Unlimited Release
Printed September 2014

Kelpie: FY2014 Project Update

Shyamali Mukherjee
Gary Templet
Craig Ulmer (PI)

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

SAND2021-1451
Unlimited Release
Printed September 2014

Kelpie: FY2014 Project Update

Shyamali Mukherjee
Gary Templet
Craig Ulmer (PI)

Sandia National Laboratories
P.O. Box 969 MS9152
Livermore, CA 94551-0969
cdulmer@sandia.gov

Abstract

The ASC CSSE project *Kelpie* is a research and development project focused on developing a distributed, in-memory data management system that can be leveraged in a number of high-performance computing (HPC) applications. After FY13's demonstration that a key/value data store could be implemented on top of the *Nessie* RDMA/RPC library, we began refactoring Kelpie in FY14 in order to make it more usable by other research teams that need it for upcoming milestones. This report provides a summary of the different efforts in FY14 that took place to make Kelpie a more usable system.

Acknowledgments

The Kelpie project has benefited from the help of a number of people at Sandia, for which we are grateful. Ron Oldfield and Jay Lofstead have consistently provided useful feedback for Kelpie in relation to how other HPC users at Sandia might benefit from this package. Todd Kordenbrock has tirelessly answered questions about how Nessie (NSSI and NNTI) operates, and debugged a number of problems we had during the implementation. John Floren from the DHARMA team committed to working with Kelpie and has helped us by evaluating whether Kelpie's APIs met his requirements. Student interns Saurabh Hukerikar and Marc Garmell worked with Kelpie and NNTI during the summer and helped test out the idea that the libraries could be leveraged in their reliable computing framework. Finally, the entire DHARMA team has provided us with a great deal of motivation to bring Kelpie to a production-ready status. We thank all of these researchers, and look forward to future developments with Kelpie in the upcoming years.

Contents

List of Figures

List of Tables

This page intentionally left blank.

Chapter 1

Kelpie Updates

Kelpie is an ongoing ASC R&D project that is tasked with developing a scalable, in-memory data store that can run on different high-performance computing (HPC) platforms and provide a flexible means by which distributed applications can share data. In FY13 we developed a prototype for Kelpie that demonstrated that with the *Nessie* communication library, we could implement a distributed, in-memory store for MPI applications that could run on both Blue Gene/P and Cray architectures, as well as InfiniBand clusters. In FY14 we have expanded this effort and have worked towards enhancing the Kelpie library to make it more suitable for application developers. This chapter provides an overview of the Kelpie library and describes differences between the FY13 and FY14 implementations.

1.1 Kelpie Overview

Kelpie is a distributed, in-memory data store that enables users to move data between compute nodes in a flexible and efficient manner. Unlike other key/value stores, Kelpie uses a high-performance communication library named Nessie [13] to orchestrate data transfers between nodes using RDMA. With Nessie's built-in support for multiple HPC network fabrics, we are able to run Kelpie on a variety of platforms, including Blue Gene (DCMF or PAMI), Cray (Gemini), and clusters (InfiniBand). Kelpie uses traditional 1D keys for most operations, but natively supports 2D keys for data aggregation tasks. Kelpie requests are all asynchronous in order to maximize network concurrency. While the FY13 prototype implemented a single distributed-hash table (DHT) on top of a collection of servers, the FY14 implementation supports a flexible architecture where users can interact with multiple pools of resources and define their own rules for managing how data is distributed among nodes.

1.1.1 Key/Value Stores

Key/value stores are a well-known means of sharing data in distributed systems. Similar to a hash table, users associate a label (or *key*) with a block of data (or *value*), and then use put and get operations to interact with the data. Key/value stores are appealing because they provide a simple abstraction that programmers can easily utilize and system designers can readily adapt to run on large, distributed systems. When constructing a distributed

key/value store, the common approach is to use a hash of a key to determine the node where a key/value pair should reside. These distributed hash tables (DHTs) are well-liked because they spread the workload across a collection of servers, and clients can determine the right node to communicate with without having to interact with a directory service.

There are many open source libraries that implement key/value stores, including memcached [11], redis [6], BigTable [8], Accumulo [1], LevelDB [10], and MongoDB [9]. Each key/value store is designed to target a specific environment, and therefore there is a great deal of variance in both what the key/value stores do and what users expect when they think of key/value stores. The main design criteria include the following:

Local vs. Distributed: While some key/value stores operate only on the local system (LevelDB and Kyoto Cabinet [5]), others utilize a collection of nodes and perform network operations to fulfill requests. It is common for developers to build a distributed key/value store out of a local key/value store by wrapping the library in network operations.

In-Memory vs. Persistent: A key design point for key/value stores is whether the intended use is to pass information between active programs or to house information persistently. In-memory key/value stores typically focus on quick interactions where data is either available in the memory of a node somewhere in the system or it is unavailable. Persistent key/value stores are focused on archiving data to disk for later use. While some persistent key/value stores are simply interfaces to storage, most use memory to cache data that is likely to be requested in the near future.

Consistency: The most challenging part of implementing a distributed key/value store is handling write consistency: what data should the store maintain when two writers attempt to write using the same key? While some systems implement traditional locking mechanisms to ensure consistent results, others relax the consistency model and guarantee that updates will resolve themselves within a specified time range. However, the most common approach is simply to avoid the problem and push consistency requirements back on the user. This approach can be effective in practice, as it optimizes for the common case where users typically follow a write-once, read-many data flow.

As a communication package that is designed to help HPC applications share data, Kelpie is characterized as a *distributed* store that is primarily focused on *in-memory* operations, with a *user-managed consistency model*. While these characteristics provide the most utility to our HPC users, it is important to note that we have not ruled out possibilities for supporting other user bases. For example, we have experimented with support for persistent storage using parallel object stores, such as RADOS in the Ceph file system.

1.1.2 Use Cases for In-Memory Stores in HPC

From our interaction with different HPC users at Sandia, we observe there are multiple use cases for an HPC-based, in-memory data store.

Workflows: Many HPC users today need to implement complex computational workflows that move data from one distributed simulation to another. These simulations codes are generally large and difficult to combine into one executable. As such, current workflows typically run each stage in the workflow as an independent job and then use the file system as a way to route data between different stages. An in-memory store would allow users to exchange data between applications without I/O penalties, and may make transfers easier to implement and debug.

In-situ Analysis: Often it is useful to be able to examine the memory of a running application to assess its state and steer its runtime. An in-memory store can provide an unobtrusive way to instrument an application and attach multiple analysis programs. Ideally, the in-memory store would maintain memory in the application nodes, in order to allow external data transfers to be performed only when they are needed.

Task-DAG Systems: A number of researchers are currently investigating the design of computational frameworks that will allow an application to dynamically schedule its work on a platform's distributed resources. An in-memory data store is valuable for this work because it provides a data plane the framework can use to manage how data is distributed in the system.

Symbolic Programming: One of the insights gained from examining task-DAG frameworks is that in-memory data stores allow users to think about their data problems at a higher level of abstraction: rather than worry about locating data and retrieving it, users simply label an object and let the data store do the work. This environment is akin to symbolic programming, and can be useful for developing complex computations in a rapid manner.

1.1.3 Kelpie Organization

As a distributed, in-memory data store, Kelpie can be configured to run as either an integrated part of a parallel application, or as a stand-alone service that end clients access. In either case, application developers instantiate a Kelpie object in their software and configure it to behave as needed. As illustrated in Figure 1.1, the Kelpie object is comprised of four main components: the Nessie communication library, a Local Key/Value Cache, a Resource Manager, and a Client Interface.

Nessie (NSSI/NNTI): All network operations within Kelpie are performed using the *Nessie* communication library, which was developed by Ron Oldfield and others at

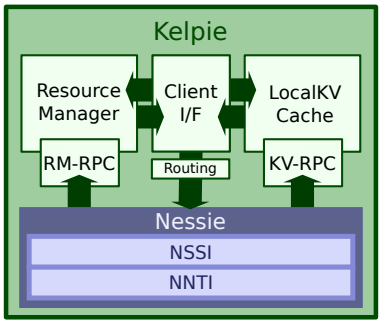

Figure 1.1. The Kelpie communication object is comprised of Nessie, a LocalKV Cache, a Resource Manager, RPC handlers, and a client interface that routes messages to the proper server.

Sandia National Laboratories. Nessie is a flexible communication library that enables users to write portable I/O services that can leverage the raw potential of modern HPC network transports. Nessie is comprised of two components: a low-level RDMA library named NNTI and a general purpose RPC library named NSSI. NNTI presents users with a generic API for RDMA, and includes custom drivers that efficiently map NNTI operations to different network technologies (e.g., Blue Gene, Cray, and InfiniBand). While NNTI is sufficient for writing network services, RDMA is a difficult communication primitive to work with by itself. It is therefore desirable to use NSSI, as it sits on top of NNTI and provides more powerful communication primitives. In addition to handling RPCs, the NSSI library performs useful operations, such as automatic memory registration, fragmentation, and argument/result packing. Compared to other RPC libraries, NSSI is appealing because it includes primitives for moving data between nodes using RDMAs. As such, RPC writers have greater control over how large amounts of data are moved between nodes.

Local Key/Value Cache (LocalKV): The LocalKV is an allocation of memory in the local application that is dedicated to housing key/value pairs on behalf of the overall Kelpie store. The LocalKV currently uses a two-dimensional hash map to organize and maintain key/value pairs. Data values are referenced through C++ shared pointers in order to minimize data copying and prevent premature deallocations. The LocalKV utilizes a set of custom NSSI RPCs to allow nodes to interact with the LocalKVs located on remote nodes.

Resource Manager (RM): The RM component is used in Kelpie to help applications maintain information about different resources that are available in the system. In many cases the RM provides a way to translate a virtual label for a resource (e.g., `/rack2/workqueue`) into a physical address NSSI can use (e.g., `ib://cn01:50901`). In addition to caching resource information, the RM provides mechanisms for automatically querying other RMs in the system to resolve queries. This approach allows Kelpie to run its resource management services in a distributed manner.

Client Interface (ClientIF): The ClientIF provides a flexible way for end applications to fetch and store key/value pairs with Kelpie. When possible, the ClientIF resolves requests using the LocalKV. Operations that cannot be completed locally are converted to RPC requests that are issued to remote nodes. In the FY13 version of Kelpie, the ClientIF simply implemented a DHT with a fixed number of nodes. In the FY14 version, users can instantiate different *Resource Handles* to dictate how requests to the ClientIF should be resolved. These handles provide a great deal of flexibility, and are described in more detail in Section 1.3.3.

1.1.4 Standard Operations Using 1D Keys

Kelpie provides an asynchronous API that is similar to other key/value stores. The API is comprised of four types of operations: `Put` inserts a key/value pair into the store, `Get` attempts to retrieve data from the store, `Drop` notifies the store a key/value pair is no longer needed, and `Info` performs a lookup to determine what information is available about a particular key. Kelpie's behavior for each of these operations is determined by the configuration of the ClientIF Resource Handle that is used to invoke the operation. While Kelpie natively uses 2D keys for all of these operations, it is expected that users will primarily utilize 1D keys for most tasks. Kelpie simply sets the second dimension (or column) of these keys to zero.

Kelpie follows the same process for resolving all four types of operations, summarized as follows. First, a user issues a new request to Kelpie by calling the corresponding function in a Resource Handle. The Resource Handle adds configuration information to the request and returns a Request Handle that the user can later query to determine the state of the operation. The Resource Handle interacts with the ClientIF to determine if the request can be resolved using information that is already available in the LocalKV. If it cannot, the Resource Handle determines which node in the system is responsible for the data, and then submits a request for the data through the ClientIF. The ClientIF serializes the arguments for the request, translates the target node's network information into an NSSI-usable address, and issues the remote RPC request to NSSI. NSSI then transports the request to the remote node for processing.

Upon receiving a new RPC message, a NSSI server will inspect the message and execute the corresponding Kelpie RPC handler that has been registered for processing the message. A Kelpie RPC handler unpacks the arguments for the RPC and consults with the node's LocalKV to determine what action should be taken. In cases where data is to be moved, Kelpie will request an RDMA transfer through NSSI. The final step of the RPC is to generate a result message that NSSI can return to the requesting node. When a user blocks on a particular Request Handle, Kelpie waits for the RPC's result message to arrive and then unpacks the message's information into the Request Handle.

In terms of moving data between Kelpie and an application, a user may reference data with either a regular pointer or a shared pointer. The distinction between these pointers is

only significant in the context of the local node. When calling Kelpie functions with regular pointers, Kelpie copies data between the application's memory and Kelpie's. This approach is easier to work with but can be inefficient as extra copies of data may be made. Advanced users may wish to instead reference data with a C++ shared pointer. In these operations, a `Put` results in the shared pointer being added to the LocalKV, while a `Get` results in the return of the shared pointer used internally by the LocalKV. While shared pointers remove copies, they can be difficult to work with, as users must be careful not to modify the contents while a transfer is in flight.

1.1.5 Aggregate Operations through 2D Keys

While key/value stores that use 1D keys and DHTs are a useful way to manage large collections of data, programmers often need a way to work on different portions of the same problem and have intermediate results aggregated together. Utilizing 1D keys to segment the data is straightforward: a user simply generates a key that encodes the necessary identifiers for each particular block of data. For example, a parallel mesh simulation may label a result generated by one compute node as "tstep-100-pressure-section-4" to designate that the data is a matrix of pressure values for the fourth portion of the mesh during timestep 100. As illustrated in Figure 1.2(a), nodes generating the data can use `Put` operations to publish their data to different nodes in the DHT. A consumer can then issue `Get` operations to retrieve all the data necessary for the next phase of work. This approach is useful when the user wants the data to remain in a distributed form. However, it can be difficult to construct higher-level services (e.g., queues or data aggregation) when components are always hosted on different nodes in the system.

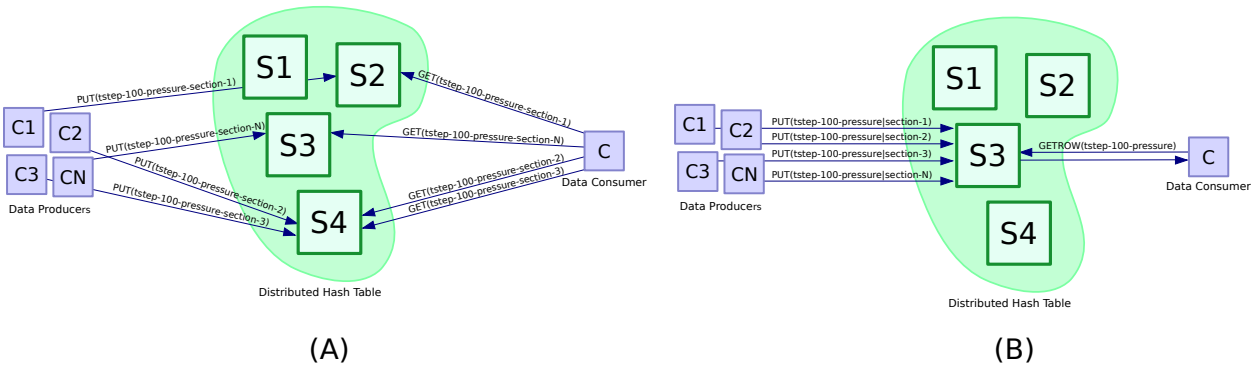

Figure 1.2. A DHT can be used to aggregate multiple results into an item that can be retrieved through a rowget operation.

Kelpie provides the ability for users to use 2D keys as a means of grouping related items together on the same server. In this approach the first portion of the key (known as the *row* ID) is used to select which node in a DHT is responsible for housing the data. The second

part of the key (known as the *column* ID) is used to ensure that different portions of data in a group do not collide. As illustrated in Figure 1.2(b), a set of data producers would use 2D keys to publish new data to the store. The row portion of the key on all operations is set to "tstep-100-pressure", while the column is set to the specific section that each producer generated. All of these operations map to the same node in a DHT because the row portion of the keys is the same for all operations. On the consumer side, a single `GetRow` operation can be used to retrieve all items within a row at the node. This operation results in a single RDMA transfer to the destination for the desired row.

The primary purpose of 2D keys is to support situations where either a number of small data values need to be combined, or a user needs to implement a custom RPC operator that requires multiple data values to exist in one location. An example of the latter can be found in higher-level containers such as work queues. An implementation of a work queue would store all of a queue's entries and metadata as different columns in the same row. In general, we expect most users to utilize 1D keys for the bulk of their work, as most users simply need a way to distribute data across many nodes.

1.2 The FY13 Kelpie Prototype

The FY13 version of Kelpie was designed to be a basic key/value store that developers could use to manage data distributed among a collection of compute nodes. Similar to other key/value stores, the user interface to Kelpie was relatively straightforward: after initializing the library, a user could issue put or get operations to transfer data into or out of the store. Internally, Kelpie managed the operations necessary for fulfilling requests if the data was not available locally. In terms of data distribution, Kelpie organized its data servers into a distributed hash table (DHT). When looking for a particular key/value, a Kelpie client would hash the key to map it to a specific server in the DHT, and then communicate with the server to complete the request. This approach is widely utilized, as it greatly simplifies the task of locating data and yields good performance when requests are diverse.

1.2.1 Limitations from a User's Perspective

While the FY13 Kelpie prototype accomplished its objectives, we found through our own assessments and conversations with other users that there were a number of ways it could be better. The following are the main limitations we found of the prototype from the user's perspective.

Lack of Peer-Based Communication: Nessie is largely oriented for client-server communication models, where a server is generally dedicated to responding to RPC requests. While this model is sufficient in examples where a pool of servers are dedicated to serving as a distributed hash table, it does not work well in cases where active nodes need

to exchange information with their neighboring nodes. This peer-based communication is fundamental to a number of data processing models and must be made feasible in Kelpie. Ideally, each node should be able to house a small, local data store and provide other nodes the ability to exchange data with it.

Namespaces: The original Kelpie utilized a single namespace for labeling data, and assumed the user would manage conflicts within the space. This approach limits how Kelpie could be utilized in scenarios where multiple applications or multiple users share the same data store and is a general security problem. Moving forward, Kelpie needs a simple way for data to be placed into different namespaces.

Opaque Data Distributions: The initial version of Kelpie focused on providing a single distributed hash table that was hidden from the user's control. We initially thought this would simplify the user's work, but rapidly found that users always had scenarios where they needed to control how data was placed or retrieved. It was clear that instead of hiding data distribution from the user, we needed to expose it and allow the user to manipulate it themselves.

Single Data Pool: Kelpie's original view of resources only allowed data to be written to a single distributed hash table. This view does not scale and does not meet the needs of complex applications that may need to support multiple pools of resources.

Flimsy Keys: The previous version of Kelpie used simple strings to label and access data entries. Users could provide one or two strings to identify the data in the sparse store. While this approach meant users could easily reference items with strings, our API doubled in size because we needed to support both 1D and 2D keys. What is needed is a more robust key naming system, where a key is an actual class.

Simplistic Metadata Management: Kelpie initially only utilized a single metadata server in the cluster to maintain the list of all nodes that were available. This service simplified how a system could be brought up and used, but was not a scalable solution. Additionally, users needed a means of storing more complex metadata about the system in order to support other data distributions. As such, we needed a better way to handle metadata.

1.2.2 Limitations from a Developer's Perspective

As the Kelpie prototype grew, we realized that the code base would become difficult to maintain if we did not start improving it. Here are the primary concerns we outlined at the beginning of the year.

Ad Hoc Testing: While we developed a number of unit tests to validate Kelpie, they were mostly written in an ad hoc manner. Kelpie therefore needed a more robust testing methodology.

No Serialization Design Pattern: The initial Kelpie utilized Nessie's XDR hooks to serialize/deserialize its data. While these operations worked, there are memory management hazards associated with using XDR data structures and C++ code. Given the availability of multiple C++ serialization libraries, we wanted to investigate how well other serialization packages perform, and develop a design pattern that would ensure data is converted properly.

Lack of C++ Standards: The source code for the initial C++ code lacked a standard coding convention. It did not uses namespaces and relied on Boost to implement features that have since been added to the C++0x and C++11 standards.

Outdated Build Environment: The Kelpie build environment worked, but was in need of repair in order to prepare Kelpie for integration into Trilinos. This work would require substantial rewrites of our CMake environment.

1.3 Conceptual Changes to Kelpie in FY14

Given the deficiencies of the prototype, the majority of our FY14 effort with Kelpie focused on overhauling the library to support a broader range of capabilities. In this section, we focus on conceptual changes we had to make in order to improve the library.

1.3.1 Improved Keys and Data Namespaces

One of the main difficulties with working with the FY13 prototype was that data values were labeled in a very simplistic way. Similar to other key/value stores, the FY13 Kelpie prototype used a simple string (or pair of strings) as a key for referencing a particular data object. While this approach is easy to use, there were two main problems with the prototype's implementation. First, each Kelpie call needed to be able to support either one or two key strings. The library used a number of wrapper functions to map 1D operations to 2D operations. Unfortunately, the multitude of library functions left many users with the impression that Kelpie was more complicated than they expected. Second, Kelpie's flat key space made it difficult to have multiple applications share a Kelpie data store due to name collisions. For example, if a user wanted to have two applications use the same data store, he or she would have to make sure that the keys used in the two applications did not conflict with each other. This task is possible on a small scale, but difficult to achieve when multiple developers want their applications to work together.

The approach taken in FY14 was to redesign the way we label data in Kelpie. First, we made a more robust key class that can work with both 1D and 2D labels. This class provides simple constructors for users to define either one or two key labels, and removes the need to define both 1D and 2D functions in the rest of Kelpie. The key class also allowed us to write

helper templates that make it easier for users to construct keys that are machine generated (e.g., row="pressure-ts-10", column="93").

Second, the FY14 uses a new "bucket" identifier to help users confine related items to a particular data namespace. A bucket identifier is a 32-bit hash value that is prepended to the row identifier of a key on all local key/value operations. It does not affect data placement, and provides a simple way to let multiple applications use the same keys without conflicts. It is not secure, but is sufficient for cooperative use provided buckets are assigned in a meaningful way. Users do not generally supply the bucket identifier on individual put/get operations. Instead, they construct a resource handle with different communication settings (such as the bucket identifier), and then perform put and get operations using those settings.

1.3.2 Flexible Communication Patterns

Another significant limitation of the FY13 prototype was that it was only designed to support client-server communication patterns, because nodes were either Nessie clients or Nessie servers. As illustrated in Figure 1.3 the client server model is sufficient to do (a) basic data interactions as well as (b) distributed hash tables (DHTs). However, nodes were unable to exchange data with their peers (c). This limitation was significant, as many users want to host data on the nodes that generate it, and then push information about its availability to other nodes.

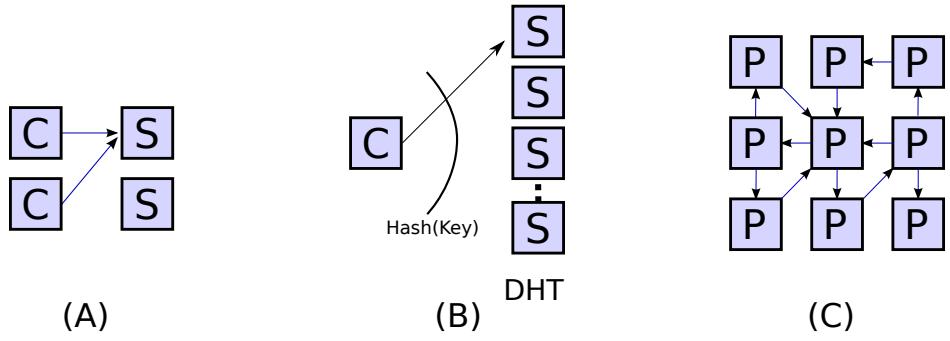

Figure 1.3. Kelpie supports a multiple communication models, include (a) client-server, (b) distributed hash table (DHT), and (c) peer-to-peer.

The FY14 version of Kelpie is more flexible and allows users to configure nodes as clients, servers, or *peers*. A peer is simply a node that has both client and server capabilities. In order to support this functionality, we updated the library to allow a node to be able to start a Nessie server in a thread. This capability provides a peer node with the ability to perform work, issue requests to remote nodes, and respond to requests.

1.3.3 Resource Handles to Resource Pools

The most significant change to Kelpie this year is in the way that resources are managed. In the prototype, Kelpie utilized a single DHT that was managed entirely by Kelpie. While this approach meant that users had a simple API that automatically took care of mapping put and get operations to a backend pool of resources, it did not meet the needs of many users for three reasons. First, the system only handled one pool of resources, which meant that users had a number of namespace and capacity problems when multiple applications used the same pool. Second, the prototype lacked a way for users to express their own custom data access policies. We frequently found that users *wanted* a system where they could micromanage how data was distributed in the system. Finally, the system had no way to create multiple *views* of different resources in the system, where each view would be customized to a specific behavior.

Our solution to this problem was to create a *Resource Handle* class that embodies all the options a client might need to interact with a particular resource in the system. As illustrated in Figure 1.4, a node in Kelpie can maintain multiple Resource Handles: one for exchanges limited to its local key/value cache, multiple handles for talking with different DHTs located in the system, and another handle for implementing a custom data distribution on different nodes. It is important to note that resource pools can overlap, and that distribution is performed by the client using mechanisms defined in the Resource Handle.

A Resource Handle is comprised of both state information and a set of APIs that are used to invoke operations. State information typically includes details such as the bucket identifier, the list of nodes used in the pool, and default settings for put and get operations. All Resource Handles implement a base set of functions for put, get, drop, and info operations that allow users to easily change out one Resource Handle for another. In the current implementation of Kelpie, there are three types of Resource Handles:

LocalKV: The LocalKV Resource Handle provides a simple way for users to interact with its local key/value cache. This resource is particularly handy when a user needs to quickly determine if multiple data pieces are all locally available without invoking network operations.

Peer: The Peer Resource Handle provides a means for a user to query a specific node to complete an operation.

Distributed Hash Table (DHT): The DHT maintains a list of different nodes participating in the resource pool and performs a key hash to determine which node is responsible for a particular data item.

Built-in handles can be generated through either their constructor or a factory, while user-defined handles can only be generated through their constructor. Future handlers that have been discussed include broadcast, multicast, reliable DHT, and a research option that stores data locally while publishing metadata to a DHT.

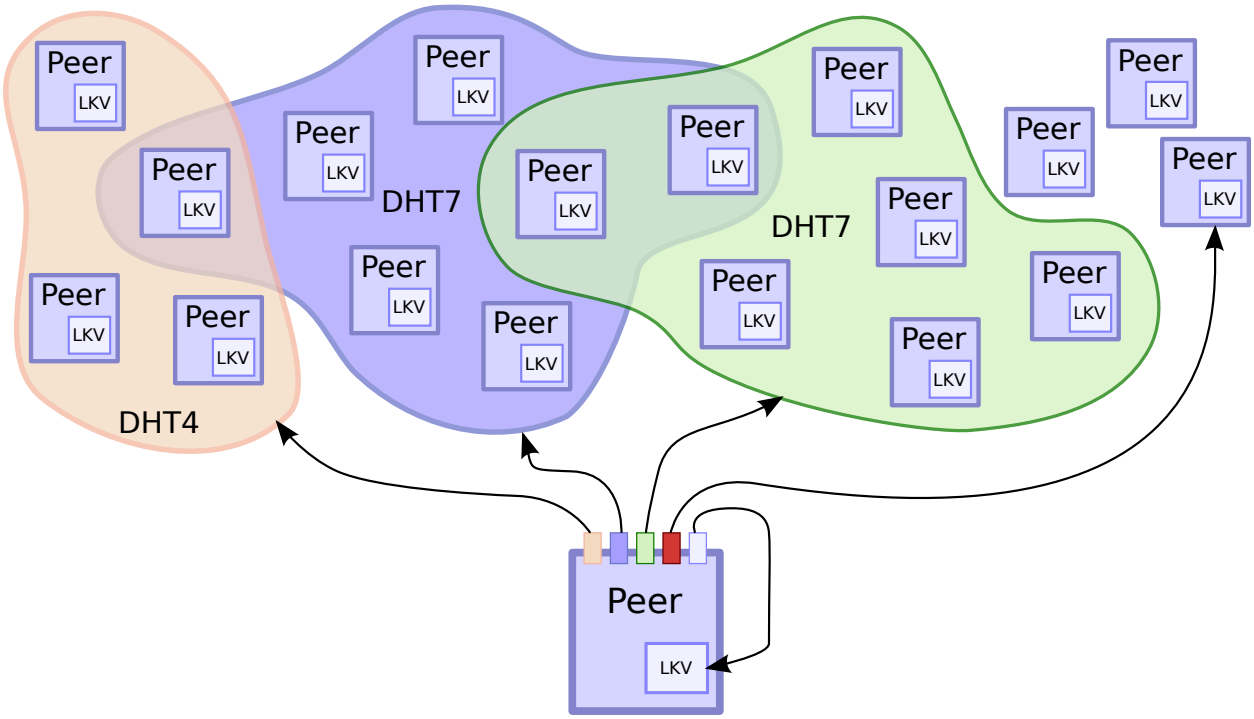

Figure 1.4. Kelpie now supports the ability to have multiple, concurrent resource pools within a single system.

1.3.4 Resource Managers

The need to support multiple resources inside of Kelpie forced us to improve the way in which we manage metadata about the resources. In the FY13 prototype, Kelpie used a single metadata server to keep track of all the servers in the system that participated in the DHT. At start time servers registered themselves with the metadata server, and all nodes would wait for the list of known servers to reach a predefined number. Clearly, this approach cannot scale because the metadata server becomes a hotspot that all nodes in the system must communicate with in order to start. Additionally, the metadata server was not designed to support the presence of multiple resource pools.

Our solution to this problem was to distribute the task of resource management to different nodes in the system, and implement a more sophisticated bookkeeping system for maintaining the nodes. As illustrated in Figure 1.5, a resource now uses a label that provides a hierarchical path in the system. Each entry in the path can be hosted on a different node, thereby allowing users to distribute management responsibilities across the system. When nodes start, they register their default path with the system as well as any responsibilities they might perform. Nodes automatically attempt to resolve unknown components in a path by querying other nodes in the hierarchy, starting with the root node.

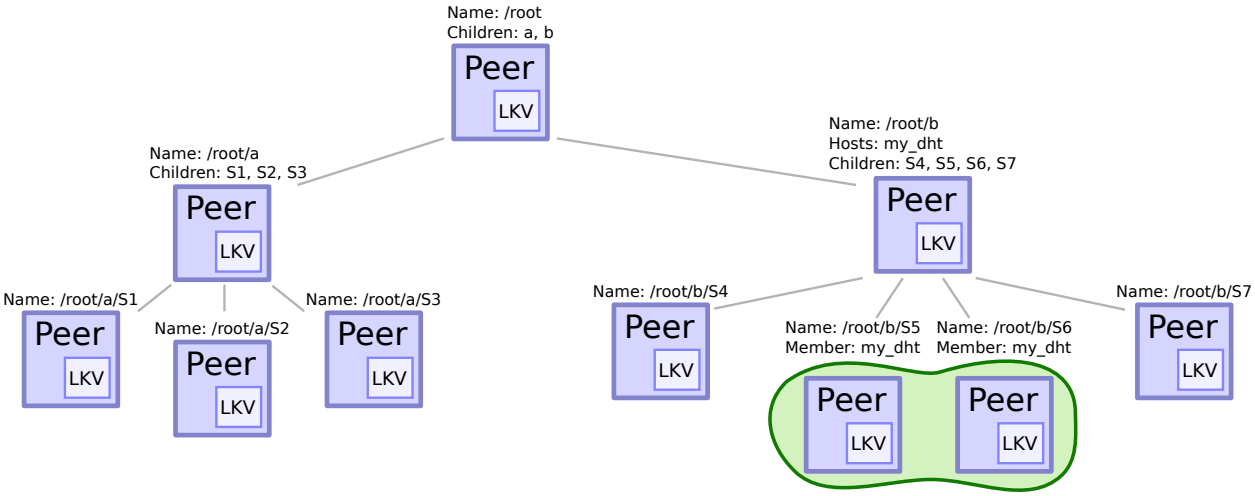

Figure 1.5. Kelpie's resource management services are distributed and provide a way to translate resource labels into runtime information.

Besides performing a virtual path to physical node translation, the resource manager helps maintain information about nodes participating in a particular resource pool. For example, a node wishing to use a DHT with the path `/root/b/my_dht` would first perform lookups to find which node was responsible for managing `my_dht`. It would then query that node to learn the identities of all the known nodes participating in the pool, as well as how many nodes are needed for it to function correctly. As Kelpie evolves, we expect this feature will be enhanced in order to allow nodes to be dynamically added or removed from a particular pool in a way that maintains order.

1.3.5 Serialization

One of the challenges of writing distributed applications that share complex data structures is finding the right way to transport data from one node to another. Data must be serialized to a contiguous buffer that can be sent as a message, and then deserialized into a data structure that the receiver can utilize. There are many serialization libraries that are available today, including XDR, Boost Serialization, Cereal, Protocol Buffers, Thrift, and Avro. Nessie utilizes XDR for its transports, and is therefore a logical choice for most Nessie programs. Unfortunately, XDR does not always map well to C++'s object-oriented programming concepts, so it is worth considering other serialization libraries.

Kelpie's RPCs utilize a number of different data structures for transporting information between nodes. In the FY14 design we investigated how we could leverage C++ classes and external serialization libraries to make the task of transferring information more straightforward. Our approach was to create each message as a class, and then provide pack and unpack operations for the class that dictate how the data should be serialized or deserialized. This approach is beneficial because it allows us to utilize whatever serialization library we want for the pack/unpack functions. In complex or hierarchical data structures we generally use Boost, while in simple structures we encode data directly.

In order to gain a better understanding of the trade-offs associated with different serialization libraries, we performed an experiment that packed a Kelpie-like message using different libraries. Details about this work are reported in Section 3.3.

1.3.6 Request Handles for Kelpie Actions

In the FY13 prototype, a user was handed back a single request handle for any operation. This handle embedded all information needed for the transaction, including the Nessie request handle and an allocation for holding result data. While this approach worked, we observed problems in a few areas. First, storing the Nessie request in the handle meant that if a user deallocated the handle before Nessie's completed the transaction, the library could corrupt data. This obligation to maintain a handle until its network operation completed was not always clear to users, and therefore users could easily get into hard-to-debug situations when a handle was not properly maintained. Second, a request handle carried a good bit of baggage with it, due to the Nessie request handle that was part of the structure. For example, transactions that could be resolved using the local key/value cache still needed to haul the Nessie handle with them. Finally, the request handle had no notion of how to resolve itself when multiple transactions were embodied in a request. This capability will be necessary in the future, as we write more complex Resource Handles that perform transfers with multiple nodes to satisfy a request.

In the FY14 we chose to decouple the Kelpie handle users see from the internal representations used to facilitate the request. The user is now handed back a simple structure with fields that designate operation completion and operation results. This handle is all that needs to be generated for requests that operate on the local key/value cache. Remote operations will result in the construction of one or more RPCRequest structures, which contain status information about each network transaction. A Resource Handle typically stores RPCRequests in a queue. When a user wants to block on the Kelpie request, the request checks the corresponding queue to determine when the actual RPCRequest has completed. When it completes, the results of the operation are unpacked into the user's handle and the RPCRequest is deallocated.

1.4 Development Improvements to Kelpie in FY14

In FY14 we took time to improve our development environment and make Kelpie more robust. These changes are briefly summarized as follows.

Google Test (Local Unit Tests): We transitioned our testing environment from an ad hoc form to one built on top of the Google Test library. This library is well known and provides an easy way to write more robust tests. The first step in this process involved adapting our existing unit tests to Google Test, as they did not require network operations to run. We found that Google Test encouraged us to write more tests, and that our unit tests served as a good place to search for memory leaks using valgrind.

Google Test (Network Component Tests): A more challenging problem was developing component tests for Kelpie in Google Test that would allow us to validate that higher-level components were also correct. Component testing is difficult in networked applications because the tests much dispatch test servers with which clients can exchange data. Our approach in Kelpie was to implement our tests as MPI programs that start a Google Test client on rank 0, and generic Kelpie servers on all other nodes. This approach has worked well, provided the tests are configured to setup and teardown the distributed environment properly. In the future we would like to construct more extensive tests that setup more elaborate network configurations.

CMake Enhancements: We revamped the entire CMake build environment for Kelpie in FY14. The build system now pulls settings from the Trilinos build in order to remove library mismatches we stumbled into in previous versions. We also added hooks to include the new testing procedures as part of the build, including automatically downloading and building the Google Test library. Integrating tests into the build has greatly helped us in our debugging efforts, as changes to either the library or a test trigger a rebuild of all the necessary components.

Additional details about how we have refined our build environment are described in Section 3.4.

1.5 Current Status

The current version of the FY14 implementation has basic functionality and is able to perform local and remote operations. However, the process of updating the software caused significant changes to both the library and test programs that have disrupted its stability. As FY14 draws to a close, we are in the process of resolving these disruptions and adapting our testing framework to more thoroughly validate its correctness. Given the changes, we have held off on signing the external release forms as well as the effort to integrate Kelpie into Trilinos's Trios package. We anticipate the code developments to be complete by the end of November, and the integration to take place shortly after that.

This page intentionally left blank.

Chapter 2

Nessie Work

Given that Kelpie utilizes Nessie to perform all of its communication operations, the Kelpie team is heavily motivated to help maintain and improve Nessie's development. In this section we discuss different efforts relating to Nessie that took place in the Kelpie project during FY14. Specifically, we have developed a new NNTI driver for BG/Q, developed NNTI and NSSI examples for new users, prototyped a named-pipe driver for NNTI debugging, and outlined a C++ version of Nessie.

2.1 An NNTI Driver for Blue Gene/Q Using PAMI

In previous work, team members from Kelpie helped port Nessie to have native support for IBM's Blue Gene/P architecture. This work was nontrivial and involved the construction of two separate transports: one for the low-level System Programming Interface (SPI) communication library, another for the higher-level, Deep Computing Message Facility (DCMF) communication library. While the NNTI drivers worked, they were built at a time when BG/P's production use was ending and BG/Q's was beginning. Given the complexities of working with SPI and DCMF, IBM opted to work towards a more advanced communication fabric when developing the BG/Q architecture. The software component of this library became PAMI: the Parallel Active Message Interface. PAMI is the successor to DCMF and is the primary, low-level communication API within the BG/Q platform. While PAMI is primarily designed for BG/Q, IBM has also ported it to run on IBM-brand InfiniBand clusters.

PAMI provides several critical improvements over DCMF that make it easier to orchestrate high-speed data transfers within a distributed application. Similar to DCMF, PAMI provides RDMA capabilities that allow a user to move data efficiently from one node to another. However, PAMI is designed to coexist with other communication libraries such as MPI or GasNet while DCMF is not. PAMI also adds support for non-contiguous data types and exposes multiple, remote atomic operators to the user. These atomics are extremely useful for notification, locking, and synchronization operations that are often needed in communication packages. Another important improvement is that PAMI is designed to be able to take advantage of hardware accelerators that are available in BG/Q. For example, PAMI leverages hardware threads, communication contexts, and wakeup regions to enhance throughput

and eliminate the need for additional complex locking mechanisms and continuous polling. PAMI guarantees all API calls execute in a non-blocking manner.

The new features in PAMI simplified the task of constructing an NNTI driver for BG/Q. As listed in Table 2.1, most NNTI operations could be implemented directly, using PAMI calls on BG/Q. This section provides implementation details for each of the NNTI operations.

Table 2.1. Frequently used PAMI functions in NNTI

Type of operation	PAMI Function
Connect	PAMI_Send_Immediate
Send	PAMI_Send
Put	PAMI_Put
Get	PAMI_Get
Remote Atomics	PAMI_Rmw
Progress	PAMI_Context_advance

2.1.1 Building NNTI functions from PAMI functions

The main challenge in adapting a new network technology into Nessie is mapping NNTI's RDMA API to the primitives provided by the network's communication library. PAMI provides all of the necessary primitives, such as memory registration and put/get operations, and utilizes a familiar callback API that can be used to facilitate transfer completion tasks such as buffer deallocation. The following summarize the approach we took to implement each NNTI operation using PAMI primitives.

NNTI PAMI Init: The goal of NNTI's init function is to initialize the transport and all of the NNTI data structures that are used to manage the interface.

1. The PAMI library is initialized through the `PAMI_Client_create` function. During this process the NNTI client is registered as a user of the PAMI library within the node.
2. The `PAMI_Context_createv` is then used to reserve one or more hardware contexts for exclusive use by NNTI. These threads ensure that the NNTI will promptly respond to network requests and that queued items will receive attention without requiring the user to poll for activity.
3. Next, we register callback dispatch routines for handling small messages received through the memory FIFO channel. The `PAMI_Dispatch_Set` function is responsible for registering specific contexts with a corresponding callback and marker. It should be noted that this function can be extremely useful as it can be used as a way to pass hints back to the user about what actions are taken during operation.

For example, a user can improve performance by registering a hint that notifies the user when an operation uses memory that is already pinned.

4. Finally, we populate NNTI's transport-specific global data structure with all the information that will be necessary in subsequent calls. Once completed, we launch a hardware thread that will asynchronously invoke the *Progress* function and stimulate outstanding actions.

NNTI PAMI Connect: NNTI's connect operation is used to establish communication between two nodes in the system, and usually takes place over TCP/IP mechanisms. Unfortunately, BG/Q's compute nodes do not have a TCP/IP stack, and therefore we cannot simply utilize traditional sockets to share connection information between nodes. Instead we must use PAMI to translate a node's identifier into a routable form, and then use primitives such as `PAMI-Send-Immediate` to help establish a connection. This operation is complicated by the fact that the function can only send small, contiguous messages, and no notification can be provided. We label these messages with a unique identifier to allow the receiver to process these messages out of band.

NNTI PAMI Register: NNTI must register any memory that is used in a data transfer in order to guarantee that the network interface is provided with memory addresses that it can utilize (i.e., usually these buffers must be physically contiguous, pinned, and translated to physical addresses). PAMI utilizes the `PAMI_Memregion_create` function to register memory.

NNTI PAMI Send: NNTI Send messages generally transport control messages that are processed by the receiver in the order they arrive. While the `PAMI-Send-immediate` function could be used to accomplish this function, we use the `PAMI-Send` function because it provides us with the ability to manage the completion of the transfer through a callback mechanism. A dispatch ID (registered at receiver side) is used to determine which callback should be used to process the message. Dispatch receive callback function implementation can specify how large payload will be received at local buffer. Current implementations like MPICH 2.2 and Charm++ deploys `PAMI_Rget` to fetch the payload packets of a large SEND buffer.

NNTI PAMI Put/Get: The bulk of NNTI's work is performed through put and get operations, which facilitate RDMAs. PAMI's `PAMI_Put` and `PAMI_Get` functions implement these operations. On the initiator's side, we utilize callbacks to handle manage buffers when an operation is completed.

NNTI PAMI Wait NNTI wait basically loops in a Progress thread which invokes a `Progress` function on a particular context until it finds the "buf-is-done" field in a work completion structure has been set. The actual work for this operation is performed in the background by a hardware thread through the `PAMI_context_advancev()` function. While this approach works, we believe we could improve performance in concurrent workloads by taking advantage of the `PAMI_context_post` function. We plan on investigating the tradeoffs between these approaches after we adapt the PAMI driver to the NNTI 2.0 API.

2.1.2 PAMI Opportunities

As researchers that have studied a number of different communication packages, we found several advances in PAMI that could be useful if they could be exposed properly in NNTI. The following represent opportunities that should be factored into future Nessie work.

PAMI Atomics: `PAMI_Rmw` provides a means of performing a broad range of remote atomic operations, such as "fetch-and-add" and "compare-and-swap" on four and eight byte data values. The atomic semantics are similar to put/get operations, and include initiator notification when the atomic operation completes. It should be straightforward for us to implement NNTI's new atomics using these operations.

PAMI Purge/Resume: PAMI provides a purge operator that terminates all outstanding transmissions with a destination. This function could be utilized to implement NNTI's new cancel operator, or possibly be leverage in higher-level communications that follow reliability protocols. A resume operation is similarly provided to restart transmissions.

PAMI Type Serialize : PAMI provides mechanisms for serializing/deserializing data to/from a buffer. It is possible that these mechanisms could be better leveraged to improve the rate at which data is serialized as it flows through Nessie.

PAMI Fences PAMI provides memory fence operations to allow users be explicit about the order in which memory transfers are resolved. These mechanisms allow a user to be notified when all memory operations to a particular fence have completed.

2.1.3 NNTI Performance on BG/Q

We have performed preliminary benchmarks with the NNTI PAMI drivers on LLNL's BG/Q Vulcan system to measure how fast NNTI can transfer data. The performance estimates for different transfer types are presented in Table 2.2.

Table 2.2. Throughput of the NNTI PAMI drivers on LLNL's Vulcan BG/Q system.

Type of operation	Throughput
Send	1.77 GB/sec
Put	1.64 GB/sec
Get	1.74 GB/sec

2.2 Developing Nessie Reference Examples

Getting started with a new communication library can be a challenging task. Every library is built with its own set of assumptions and it is often difficult to piece together just how the API should be used to perform common operations. When we first started using Nessie, we were overwhelmed. While Nessie has good doxygen documentation and a few detailed examples, it lacked tutorials that new developers could use to walk through getting started with the library. During the Spring, we were faced with the problem of getting a pair of Summer students working with Kelpie and Nessie so they could complete exploratory tasks in the Dharma project. We decided to go about building a set of basic examples for NSSI and NNTI that would help illustrate the different ways the libraries could be used. In the process of writing these examples, we discovered a few stumbling points that helped us identify problem areas that could be improved in the libraries.

2.2.1 NNTI Examples

Our Summer students were tasked with writing an efficient, collective heartbeat algorithm for Dharma that took advantage of RDMAs. Given that this work did not need RPCs, the interns focus on working directly with NNTI. In order to help them get started, we developed a few examples that illustrated the process of registering memory, connecting to other nodes, and transferring memory. Currently, there are two main C++ examples constructed in our repository. Both of these examples launch a set of client/server nodes using MPI, and then transfer data between the nodes.

Simple Put: Our first example provides a client class and a server class, and simply uses a Put operation to move data from the client to the server. This example shows how to manually register blocks of memory with NNTI and share the structures using MPI operations. The intent of this example is simply to show the minimal number of operations necessary to setup communication, and to illustrate how C++ classes can be used to compartmentalize data that NNTI uses.

Timing: A common question we find users asking is, *how expensive are different communication operations in NNTI?* To help answer this question, we created a set of timing examples that setup a base environment and perform an operation multiple times. Currently we focus on memory registration operations, putting blocks of memory, and sending short messages. In the future we hope to expand this work to include higher-level communication patterns.

We had a few stumbling points when writing these examples that were caused by our own misinterpretation of NNTI. First, the Simple Put example was constructed with the assumption that the receiver of a put operation would be notified when the buffer received an operation. While this functionality used to exist in older versions of NNTI, it has been

removed because it is unclear how the library should respond when multiple puts take place to the buffer. Unfortunately, the MPI transport still implements the functionality, so when we switched to using InfiniBand, we were stumped as to why acknowledgment never happened. While this acknowledgment can be re-enabled through an environment variable, we decided it was more desirable to uniformly disable it on all transports and instead use send commands to notify.

A second problem we had during the timing example was that our testing framework got into a deadlock scenario during initialization where neither side could make progress. Todd Kordenbrock realized that the problem was our client and server were both attempting to connect with each other at the same time. Unfortunately, the connect operation goes into a blocking operation that does not allow the application to re-scan the status of its connection socket to see if there is new work. The client/server model for NNTI does not normally exhibit this problem because servers always poll for new connections while clients do not expect to receive connections. We currently do not have a foolproof solution to this problem. One way these conflicts can be avoided is to devise a connection schedule where nodes take turns establishing connections to each other.

2.2.2 NSSI Examples

We were motivated to build a set of NSSI examples for multiple reasons. First, Kelpie directly uses NSSI and therefore there have been several times in this project where we wanted to perform basic experiments with NSSI to make sure it would behave as we expected. Second, we wanted to document some of the different ways to do things in NSSI so that we could revisit them if needed. Finally, we have discussed building a C++ version of NSSI that would be built from the ground-up to support C++ code. It is therefore extremely useful to have a set of examples that we could test against to validate that a new version of NSSI does not break existing capabilities. The NSSI examples we developed in FY14 include the following.

Build Trilinos: A common problem for new users is simply figuring out how to download and build Nessie. This example provides instructions on how to obtain Trilinos, and uses a configure script to create a Trilinos build with the minimum set of packages necessary for Nessie.

Trampoline: Older versions of NSSI did not have support for RPCs that were embedded in C++ classes. This example illustrates how a simple C function and a global variable can be used to trampoline into a particular class. It is recommended that users instead use the newer API, which provides the ability to register RPCs inside of classes.

Simple: Simple is a basic example embeds NSSI initialization operations into a class in order to make it easier to write simple clients and servers.

Client-Server: The client-server example expands on the simple example and utilizes individual client and server classes to implement network behavior. This approach also uses

the new C++ doRPC mechanism and illustrates how a single class can host multiple RPCs. In addition to providing an all-in-one example that is launched through MPI, InfiniBand users can build an executable that can be launched by hand on different nodes.

Alternate Serialization: It is often desirable to use C++'s object-oriented nature when serializing data from complex data structures. This example illustrates how a user could use a C++ serialization library such as Boost to encode/decode data, and then pass it into a simple XDR structure that NSSI can utilize.

Timing: The timing directory is a placeholder for examples that will be used to measure timing aspects of NSSI. We currently provide a simple ping example, but expect to construct others tests in the future.

2.2.3 Example Availability

The NNTI and NSSI examples are currently hosted in a separate software repository from the Trilinos repository where Nessie resides. At a later point in time, we would like to refine these examples and make them available as part of the full distribution.

2.3 Investigating a C++ Version of Nessie

Kelpie development from time to time has been hindered by the fact that Kelpie is written in C++ and Nessie is written in C. While C and C++ are similar enough that the languages should work together, there have been times where the language differences have caused problems. For example, the fact that C and C++ handle strings differently has caused memory management problems with Kelpie when dealing with XDR due to string conversion issues. These mismatches have made us wonder whether if our problems would be simplified if we could remove some of these transitions. While it does not make sense to change NNTI, we believe there would be multiple benefits for converting NSSI to C++. In addition to providing a better fit to Kelpie's needs, a C++ version of NSSI could offer an opportunity to enhance NSSI's capabilities. As such, we began investigating how to transition NSSI to C++.

2.3.1 Planning NSSI Improvements

Doing a complete rewrite of NSSI was outside the scope of our development goals for FY14. As such, we instead focused on analyzing what bothers us the most about the NSSI interfaces when used from C++. The main concerns include the following.

Global Variables: NSSI utilizes a number of global variables to keep track of state. C++ offers an opportunity to encapsulate these items into a MonoState class that stores the globals as static data members. While these variables are still technically global, they are contained within a single class and are easier to identify.

Namespaces: With C++ Nessie variables and functions can be scoped using a namespace, such as "trios::nssi". Namespaces help group related items together and can help eliminate the need to prepend long function names (e.g., "trios_nssi_foo" becomes "trios::nssi::foo", or simply "foo" when working in the "trios::nssi" namespace).

Grouping: C++ offers an opportunity to group related functions together into a class. The Kelpie project instantiates its own custom class to hold NSSI operations (e.g., NssiCore), and it helps us separate network operations from other parts of Kelpie. It is logical that NSSI should natively provide similar groupings.

Containers and Strings: C++'s provides a number of robust containers for maintaining data. NSSI already uses some of these containers internally, but does not usually expose the structures in order to be compatible with C programs. Using C++ as the base language would allow us to leverage these structures throughout.

2.3.2 Early Work

As a first step towards transitioning NSSI to C++, we began an adaptation that took the existing API and wrapped it into a C++ form. This work was intended to be a refactoring of NSSI and not a complete rewrite of NSSI. The main goal was to eliminate or minimize the number of global variables as they present difficulties in testing. While good progress was made in this endeavor, we ran into two challenges that stalled our effort. First, we found it difficult to validate our changes because we lacked NSSI examples that demonstrated expected behavior. The creation of the NSSI examples this year should be sufficient to resolve this problem. Second, the NNTI interface changed this year, requiring updates to the original NSSI code. Rather than go through the adaptation process twice, we decided to postpone the work and concentrate on other tasks. Our first draft of the C++ version of NSSI is stored in our repository, and will be revived in the future when needed.

2.4 Restructuring of Trios

As Kelpie approaches a state where we feel comfortable releasing it externally, we are investigation ways to incorporate it into the Trilinos library. Since Kelpie is tightly coupled with Nessie, it logically belongs with Nessie in the Trios package. However, there was a question as to where exactly it should be placed. Should it be a child of Nessie? A peer to Nessie? or perhaps just a service in Trios? The more we looked at the layout of Trios, the more we began to wonder if a broader reorganization was not in order.

Our investigation of a restructured Trios is aimed at bringing a more intuitive and well-defined organization to the Trios sub-packages. Currently sub-packages are not clearly defined. For instance, the top-level "examples" directory contains mainly NSSI examples, but it is not clear from the directory structure that this is the case. We feel a better solution would be to clearly define Trios sub-packages, making it clear that they can be built separately, and wholly containing their own examples, tests, and tutorials. The current Trios approach to specifying dependencies would be maintained. We are not rewriting Tribits, just refactoring the way Trios is currently using CMake and Tribits. We aim to make Trios look like it's peer Trilinos packages. In particular the SEACAS sub-package is a good model for Trios as it handles many sub-packages of it's own. Specifically, our goals are to:

- Make it easier for the inclusion of new trios sub-packages.
- Make it easier for new users and developers to incorporate Trios and it's sub-packages into their projects.
- Clearly define locations for tests, examples, and tutorials within a Trios sub-package.
- Make it clear that Trios can mostly be regarded as an "umbrella" package for its sub-packages. Specifically, there would still be a libtrios.a produced that contains tools helpful in maintaining Trios sub-packages.

This work is in conjunction with and informed by developing a CMake build for Kelpie. From that effort we learned that making Kelpie easier to build is directly correlated with developers being more willing to use it. Many researchers and programmers are unfamiliar with CMake and tend not to trust it since it does not look like a build system they are familiar with. Our experience is that users unfamiliar with CMake have a difficult time with it unless the build/link process is elementary. We aim to make building Trios (and Kelpie) as easy as possible.

2.5 A Named-Pipe Transport for NNTI

One of the challenges in debugging NNTI code is tracking the progress of different operations as they move through the communication fabric. Currently, we debug NNTI programs by running them first on a development computer, using MPI or Portals as the transport. While these transports make it possible to run examples with multiple ranks on a desktop, they can sometimes get in the way of the debugging process. For example, in NNTI's MPI transport operations may conflict with the higher-level MPI operations a user is doing at the application level and cause unexpected bugs. Another problem is that users generally do not have access to the actual messages being shipped over MPI. This condition makes it difficult to "see" the low-level network behavior of the system (e.g., in a manner akin to using *wireshark*).

In order to provide better debugging support, we investigated whether it would be possible to implement a *named-pipe* transport for NNTI. The idea was simple: make an NNTI transport that connected to other local processes through UNIX named pipes. As illustrated in Figure 2.1, each NNTI task would connect with a switchbox task that would be responsible for routing data to the proper node in the system. The advantages of this approach are (1) we could get a better handle on how messages are flowing through the system, (2) we could emulate how programs start/stop at different times, and (3) a user could run NNTI applications without using MPI.

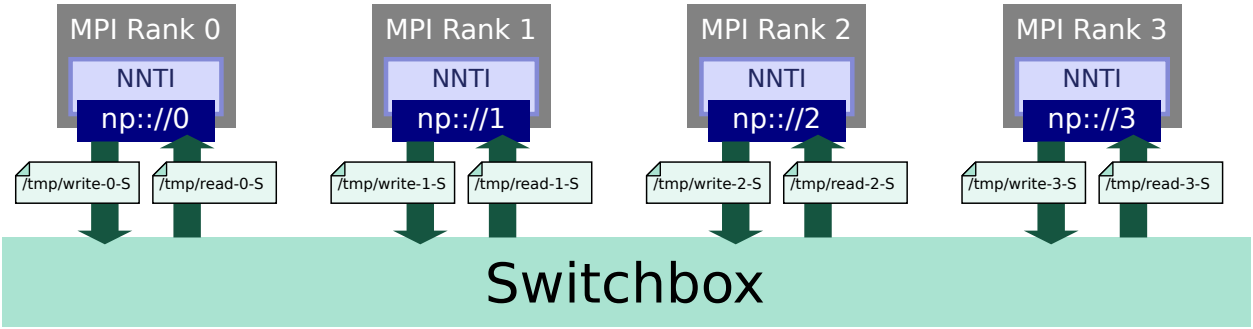

Figure 2.1. For a named-pipe NNTI transport, ranks pass data through named pipes and use a switchbox process to route messages.

2.5.1 Named Pipes

As a review, a named pipe is a simple IPC construct in UNIX that allows a user to send data from one process to another through semantics that are part of the file system. A named pipe appears as a file and supports one reader and one writer. By default, a reader will block if it attempts to read data from the named pipe and it is empty. However, a writer will not (normally) block when writing data into the named pipe because the kernel allocates additional memory to buffer data inserted into the pipe as needed.

A named pipe by itself does not provide a useful way to debug an application. However, it does provide an easy way for us to chain different processes together in a UNIX environment. In the simplest case we could use a named pipe as a way to capture data (or messages) generated from one process. By writing more complex programs that read from one set of named pipes and write to another, we can simulate a real network that provides us with full visibility into what messages are in-flight at any given time.

2.5.2 Named-Pipe NNTI Stub

The first step in implementing a named-pipe transport was to construct an NNTI driver that could read and write to a pair of named pipes. Given that we did not want this debug code to interfere with existing transport code, we decided to use environment variables as a way to pass a node the names of the named pipes it would use to send and receive data. We then constructed functions for the driver that implement each of the network operations NNTI performs (put, get, and send). A simple data format was chosen for encoding these operations in order to make it easier to write the switchbox and any other analysis modules that would be added to the system.

2.5.3 Switchbox

In order to route traffic from one process to another we developed a switchbox process that monitored all named pipes that were written to by the NNTI drivers. This task monitors the list of incoming pipes, extracts the next incoming message, and then routes it to the proper destination. Rather than use non-blocking I/O we simply wrote the switchbox in as a multi-threaded application, where each thread monitors an incoming pipe.

2.5.4 Challenges

A number of challenges arose from this work.

Bi-Directional Issues: Pipes are easy to manage when data is moving in one direction. However, the NNTI driver has both incoming and outgoing data, and a requirement that the driver not block for long periods while doing its work. We discovered multiple chicken-and-the-egg scenarios where incoming and outgoing interactions could not make progress.

Pipe Capacity: Pipes have a PIPE_BUF limit to how much data can be passed into the pipe at a time. We found that this limit conflicted with the size of the RDMA messages we wanted to transport.

Difficult Setup: While the setup could be scripted, we found it took a lot of pipes to route data in the scenarios that were of interest to us. It would be worthwhile to establish a more robust way of constructing all of the pipes.

2.5.5 Status

In this experiment we modified the NNTI code base to support named pipes and constructed simple scripts to create the necessary set of pipes for a simulation. While the implementation

supports send, put, and get, it does not support wait as of yet. While we feel this has been a useful experiment for us, we do not plan to integrate the work into the main repository, as the overhead for getting a working environment running is more costly than the benefit we believe we get out of the system at the moment.

Chapter 3

Experimental Tasks

In order to grow Kelpie into a more mature product, we feel obligated to look into ways that we could leverage other technologies or otherwise do things differently in Kelpie. This chapter describes experimental work we were involved with this year that explores and prototypes different ideas that we expect to need in the near future.

3.1 Leveraging Ceph for Persistent Storage

While Kelpie is primarily designed to support in-memory storage, many users would benefit if Kelpie could also interface with a persistent data store. This capability would allow users to warehouse data for later use, and could be used by Kelpie as a staging area for data that is evicted from memory. In the FY13 Kelpie prototype we explored ways by which we could utilize a standard POSIX file system as a way to offload Kelpie data. This year we began exploring how existing HPC file systems such as Ceph could be utilized as a way to house Kelpie's objects.

3.1.1 Ceph and RADOS

Ceph is a parallel storage system for clusters that was initially developed by Sage Weil for his doctoral dissertation [15] at UCSC. Ceph is designed to be a *distributed object store* upon which additional storage services are layered. While most users are more familiar with Ceph's file system and block device services, users can also use the librados library to interact with Ceph's native object store, named RADOS (Reliable Autonomic Distributed Object Store) [16]. RADOS decomposes objects into smaller blocks that are replicated to multiple object storage devices (OSDs) in the cluster. RADOS keeps track of where different object blocks are stored in a cluster through metadata servers (MDSs), and uses a hashing algorithm called CRUSH to locate a block within a collection of OSDs. This approach is beneficial because a client can retrieve the list of MDSs associated with an object when it is opened, and then use the information and the hash algorithm to locate data blocks as it references them.

The librados API has many distinguished features which make it an ideal choice for use with Kelpie as a backend, persistent object store.

- RADOS provides independent and balanced object placement across a dynamic heterogeneous storage cluster. Similar to Kelpie, RADOS provides a simple interface that maps well to parallel resources.

- RADOS ensures a consistent view of the data distribution, as well as consistent read-/write access to data objects through the use of a versioned *cluster map*. Every storage node therefore has complete knowledge of the distribution of data in the system.

- A user can specify special attributes that are stored along with each object. This trait enables a developer to associate application-specific metadata with an object without having to make it part of the data component of the object. Additionally, these attributes can be used to refine queries when searching for objects in the librados API.

- Objects can be set to have an object map consisting of multiple, variable-sized key/-value structures. This feature enables a developer to pack multiple items into a single RADOS object. Because RADOS manages these containers, users can easily add key/-value pairs to the object, or narrow their queries to specific keys.

- Multiple read or write operations on a particular object can be scheduled to be performed asynchronously. This trait enables a developer to overlap requests and improve performance.

- RADOS permits object cloning, deletion, and modification to take place on fine granularities. This characteristic provides developers a great deal of freedom to express complex operations, with the understanding that the system will take care of resolving conflicts.

3.1.2 Prototyping a Kelpie Interface to RADOS

As a first step towards constructing a Kelpie interface to RADOS, we constructed a stand alone application that stores and retrieves Kelpie-like key/value pairs. During this process we realized that there are multiple ways we could organize data that is stored in RADOS, and therefore it would be beneficial to conduct experiments that investigate the trade-offs of different approaches. We began by constructing a general storage class that defined the programming interface Kelpie would use to read and write objects. We constructed test programs to generate synthetic key/value data using this interface. We then implemented two different derived classes for the interface that used different strategies for organizing the data in RADOS.

FlatStore: Our first strategy for organizing data in RADOS was to map each 2D Kelpie key to a single RADOS object. This process effectively flattens the address space by concatenating the key's row and column identifiers together into a single string. The advantage of this approach is that each cell of data in Kelpie is placed into its own

container, and therefore it is straightforward to store and retrieve individual key/value pairs. The disadvantage of this approach is that data is not grouped together inside RADOS, and therefore users must issue multiple requests to retrieve a row of data,

IndexStore: Our second approach was to utilize RADOS's internal key/value pair capabilities to group related items together. In this approach, we use the row portion of the Kelpie key as the RADOS object name, and then use the column portion of the key to identify different portions of the row inside the object. This approach is advantageous because it groups related items together and allows row operations to complete with fewer remote operations. The disadvantage of this approach is that its performance is dependent on how well RADOS performs its key/value operations.

3.1.3 Early Experiments

We implemented both data interfaces and performed initial experiments on a single desktop to validate that RADOS performed as we expected. Our initial trial created 1,000 Kelpie objects of varying sizes that were then written and read back from the RADOS store. We made the following observations from this experiment.

- In both FlatStore and IndexStore, it is advantageous to fill in an object's data values and attributes at the same time, as opposed to writing the values and then modifying its attributes later. This trait is especially true in IndexStore, where multiple values are stored to the same object. This observation is not surprising, as every transaction with RADOS incurs its own locking and synchronization overheads.
- IndexStore always performed better than FlatStore in terms of timing and throughput. This trait is due to the fact that it reduces the number of interactions that are required to complete the row operation.
- RADOS naturally performs better when it is provided more data to work with in a single transaction. FlatStore and IndexStore both offered better performance as data size increased, plateauing at 16KB. IndexStore also improved as the number of columns per row was increased, plateauing at 64 columns per row.

From these observations we expect that the IndexStore approach is a better way to implement the data store on RADOS. We will continue this investigation in the near future when we move towards integrating Ceph storage into Kelpie.

3.2 Enabling Multiple Serialization Strategies in Nessie

Nessie is designed to use the well-known XDR library to serialize and deserialize data transported over the network. This approach has worked well for nearly all our C examples: a

user defines the schema for data structures that are to be transported and XDR generates C code that (packs/unpacks) the user's data (to/from) a contiguous buffer. XDR is well-tested and simplifies the task of debugging and memory-leak checking RPCs.

Unfortunately, XDR can be cumbersome to work with in C++ because it uses plain C structures to house data. Memory management can be a problem for fundamental data types (e.g., `string` vs. `char *`) as well as in raw allocations (e.g., `malloc` vs. `new`). C++ users may also choose to avoid XDR because it is often desirable to express additional operators for a data structure that is being transported. These operators can be difficult to merge in with XDR's autogenerated code stubs.

In order to make Nessie more appealing to general users, we explored ways in which we could layer other serialization mechanisms on top of XDR. We wanted to provide the most C++ flexibility while minimizing the process of effectively serializing the data twice. We then focused on capturing the data flow as a design pattern that we could easily reuse throughout Kelpie.

3.2.1 Minimizing XDR Work

The first step in making this process was to find a way to send a variable amount of data through XDR without adding significant overheads. This task is complicated by the fact that XDR does endian operations and data padding to make its encoded data universally readable. For example, a naive approach of simply declaring an XDR container with only a variable-length char record is ineffective, because XDR may align each character to be on a 32-bit word boundary. Similarly, using a string container fails for non-text data because the string cannot hold binary data.

```
/* A blob for XDR  */
struct transport_blob {
        opaque blob<>;
};
```

The solution is to use a variable length *opaque* field. The opaque setting tells XDR not to endian-fix the data or pad it. As such, a user with data that is already serialized may set the pointer and length of the field and be assured that XDR will simply pack the length field and copy the opaque data.

3.2.2 Moving from C++ Structures to XDR

The next step in utilizing different serialization libraries was to devise a simple standard for orchestrating the serialization/deserialization process that could be directed by Kelpie. Our approach to handling this was to construct a base class to inherit from that required the definition of two functions: `Pack()` and `Unpack()`. Users are free to utilize whatever

serialization functionality the want as long as these functions convert the class's data to and from a binary string that can readily be transported as an XDR opaque blob.

We constructed serialization helpers to simplify the amount of work required to convert data between formats. Given that Boost is one of the most widely-used serialization libraries, we constructed functions `BoostPackClass()` and `BoostUnpackString()` to invoke Boost's serialization handlers properly. Similarly, we created helper functions `ConvertBlobToString()` and `ConvertStringToBlob()` to convert data to our XDR opaque blob format.

```
class MsgExample : public Message {
  public:

     uint8_t flags;
     vector<rm_resource_t>  resources;

     MsgExample() : flags(), resources() {}

     //Boost's serialization/deserialization methods
     template <typename Archive>
     void serialize(Archive &ar, const unsigned int version){
       ar & flags;
       ar & resources;
     }

     //Pack this class into a string
     string Pack() const {
       return BoostPackClass<MsgExample>(*this);
     }

     //Use Boost unpack a string to this class
     void Unpack(string const &s){
       BoostUnpackString<MsgExample>(*this, s);
     }
};
```

Figure 3.1. A serializable message can be defined by inheriting from the Message class, implementing the Pack and Unpack functions, and defining a serialize function that Boost can use.

Figure 3.1 provides an example of how a serializable message can be defined that will work with Kelpie. This example uses Boost to perform serialization. The `serialize()` function performs both serialization and deserialization in Boost, and simply involves adding all necessary variables to an archive. As this example illustrates, containers such as vector are handled by Boost, provided the items in the container are also serializable. The `Pack()` and `Unpack()` functions simply utilize our serialization helpers to convert between the class and a string. Not shown in this example is how data is converted to XDR. Kelpie handles this process, using the helper functions that convert between a string and an opaque blob.

3.3 Comparing Serialization Performance

In order to better understand the features and performance differences of today's popular serialization frameworks, we conducted an experiment that measured how long it took to serialize a Kelpie-like data structure using different libraries. In addition to performance comparisons, this experiment gave us development experience with other serialization libraries and has helped us evaluate whether the libraries would simplify Kelpie.

3.3.1 Serialization Libraries

There are a number of open-source serialization libraries available today for C++ that make it easy for programmers to marshal data and make it ready for network transport. These libraries largely stem from the fact that C++ lacks built-in support for both automatic serialization and portable introspection. The most popular open source packages for accomplishing serialization in C++ are as follows:

External Data Representation (XDR): XDR is a serialization package that was developed by Sun Microsystems in the 1980s. Users specify their data structures in an external file and then use command line tools to generate C code that packs and unpacks the data structures using calls to the XDR library. XDR can support complex structures with nesting, provided that all the structures are defined in XDR. A criticism of XDR is that serialized data is converted to a big-endian format with 32-bit alignment. This approach can lead to poor packing and scenarios where arrays of words must all be endian converted if the designer is careless in defining his or her data structures.

Boost: Boost's serialization is a well-used package for encoding and decoding classes in a flexible manner. A class can be made serializable by adding a single serialize() function to the class that can be used for both encoding and decoding. This approach makes it easy to integrate serialization into a class, and supports the ability to nest items as components are processed one-by-one in the data stream. Boost can support multiple types of intermediate data products. While text- and xml- based transports are often utilized, the library supports a binary version that packs bytes efficiently.

Cereal: Cereal [4] is a "headers-only" C++11 project that is designed to provide a lightweight serialization alternative to Boost. Cereal's API is very similar to Boost's and therefore it is often straightforward to transition existing code to use Cereal. One drawback of Cereal is that it requires C++11, which is not always available on older production systems.

Protocol Buffers: Protocol Buffers is an object-oriented serialization library developed by Google that focuses on efficiency. Similar to XDR, it generates source code from a definition file. Protocol Buffers differs in the way that users interact with the generated

code. Instead of simply populating data structures that are later serialized, users call getters and setters on the generated classes to modify the serialized object's state. The advantage of this approach is that the object can track which fields are populated and use the information later in the serialization process. Protocol Buffers has a reputation for being both fast and space efficient.

There are other serialization libraries we looked at but did not have time to examine thoroughly. Apache Thrift [7] is widely used in the networking community because it has support for many languages. Thrift was developed by Facebook and provides both serialization mechanisms and RPC operations for servers. Apache Avro [2] is a newer alternative to Thrift, and has a reputation of providing good performance in general tests. The primary author of Protocol Buffers version 2 has developed a library called Cap'n Proto [3] that focuses on performance optimizations that go beyond what Protocol Buffers can currently support.

3.3.2 Synthetic Tests with Kelpie Data Structures

Our goal in this task was to gain a better understanding of how different serialization libraries performed using data structures similar to the ones used in Kelpie. In this effort we constructed a simple reference message containing multiple default values and a variable number of id/url data entries. We implemented a simple benchmark that generates a number of data structures, and then goes through the process of encoding/decoding each item with different serialization libraries. We measured the average number of bytes used to house each message, as well as the message encoding/decoding times.

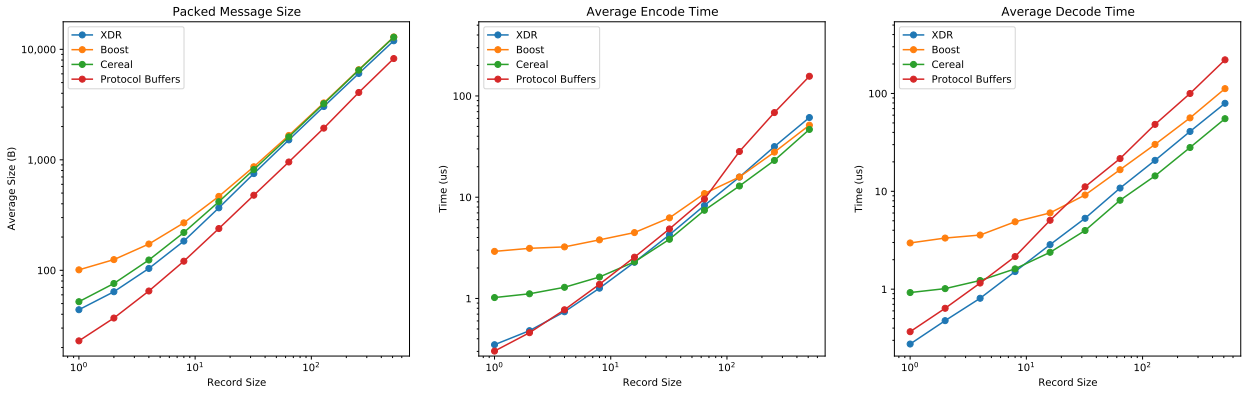

Figure 3.2. Performance comparison of different serialization libraries when encoding a data structure similar to the ones used in Kelpie.

The performance measurements for the experiment are presented in Figure 3.2. In general, we are most interested in the left side of the plot, as Kelpie messages do not typically

employ a large number of records. In terms of encoded data sizes, Protocol Buffers consistently provided smaller messages, followed by XDR. For small numbers of records, XDR and Protocol Buffers had much better encoding/decoding times than Boost or Cereal. However, Cereal did offer the best performance after 16 records of data. In general, Boost gave the least favorable results, with slow encoding/decoding times and the largest message size of all the systems.

3.3.3 Impact on Kelpie

Kelpie currently uses Boost for most of its message formats, because it provides the least-invasive way to implement serialization in a C++ manner. A logical next-step would be to transition Kelpie to use Cereal, or possibly Protocol Buffers. While these libraries offer advantages, it is unappealing to switch them out now because they make the build environment more complicated. Non-standard library dependencies and C++11 make it difficult for new users to build the project. It may be more practical to revert back to using XDR, and deal with the C to C++ mismatches a we have done in previous implementations.

3.4 Refining Kelpie's Build Environment

One of the main hardships new Kelpie users have had to deal with is understanding all of the finer points of our build environment. In the FY13 implementation, Kelpie utilized a basic CMake build process that included a number of basic unit tests that ran in an ad hoc manner. When the user's environment was configured properly, Kelpie would build and individual unit tests could be used to test out the functionality of the library. Unfortunately, when the environment was not configured properly, it could be difficult to determine why the unit tests were failing as well as what was wrong with the environment. This year we experimented with different ways that we could refine the build environment and make it more accessible to our users. We achieved these improvements by improving Kelpie's CMake code and exploring the benefits of Google's Testing library.

3.4.1 Improving Kelpie's CMake Environment

The first step in improving Kelpie's build environment was to refine how the library worked with CMake. CMake is a cross-platform build system that utilizes configuration files to help generate Makefiles that can be used to build a library on other platforms. CMake makes it easier to express macros for building different components, and provides a convenient user interface for viewing and editing different aspects of the build's configuration. CMake is designed to be an *out-of-source* build system that allows the user to build a library in a directory tree that is different than the source code's directory tree. Trilinos utilizes CMake,

and therefore Kelpie will need to have a CMake-based build system when it is integrated into Trilinos.

Currently, Kelpie is built as a stand-alone library that is outside the Trilinos distribution. While maintaining Kelpie as a separate library makes Kelpie development easier at times, it also complicates the build process because Kelpie does not automatically inherit all the build settings that were defined in the Trilinos build process. As a result, users of the FY13 Kelpie prototype often misconfigured their builds and received unexpected errors. In FY14, we went to great lengths to improve Kelpie's CMake environment. We now pull a good number of configuration settings from the Trilinos installation, and throw build errors with meaningful messages when there are conflicts. The lesson we learned in this process is that it is important to catch incompatibilities as early as possible.

We also took the opportunity while restructuring our build environment to better organize both our CMake code and the way we build our library components. We chose to migrate a number of CMake helper files out of the top-level source tree into a common location to clean up some of the clutter that developers found distracting. We then changed the way the system built. Previously, library and test code built independently of each other. This helped separate the two, but ultimately made development tedious. Recently, we have adapted the build environment to allow both the library and testing parts of the build to be part of the same system. This approach enables us to be assured the changes to either the source or the testing code trigger a rebuild of all the necessary components.

3.4.2 Leveraging Google's Testing Library

Another significant improvement in Kelpie's build systems was to switch to using Google's Testing library for all our code testing. Google Testing, of GTest is a well-known library for C++ that makes it easy to write testing operations. We transitioned a good bit of our testing code to utilize the GTest code and found that it created much more uniformity in Kelpie. Another useful feature of GTest is that it is easy to incorporate into CMake builds. We updated our CMake files and defined mechanisms that will download missing GTest software libraries and built them if they are not available locally.

One hardship in debugging network code is developing tests that allow a user to check the behavior of network operations. We experimented with GTest and found that while it did not provide any special tests for client/server interactions, we could construct our own tests using MPI. For these tests we generally use MPI to launch a client and multiple servers, and then invoke GTest operations only on the client. While this is less than ideal, we have found that the testing covers a number of important scenarios we care about.

Looking forward, we anticipate adding more test programs to Kelpie that use GTest to validate correctness. The current build environment provides us with a useful testing framework that we can extend to implement new tests.

This page intentionally left blank.

Chapter 4

Integration Approaches

In addition to developing the Kelpie library, we are also actively exploring how in-memory data stores could benefit the HPC community. This investigation is an ongoing process that continues to evolve as Kelpie matures and we learn of more scenarios where SNL HPC users can use Kelpie as a better way to manage distributed data. Given that Kelpie provides a rich environment for data operations that could be utilized in a variety of ways, the research question in this work is *how can we integrate Kelpie into HPC codes and leverage its capabilities?* Kelpie integration can take place at different levels, depending on how much effort developers want to invest in updating their codes. While developing Kelpie, we explored three different approaches to integration: *(i) workflow-based integration* techniques that focus on the task of moving data between legacy applications without significant modifications, *(ii) library-based integration* techniques that switch out existing communicators for ones that have been customized to work with Kelpie, and *(iii) custom data-plane integration* techniques that utilize Kelpie's native interface to implement an application's data flow.

4.1 Workflow-Based Integration

The first way that Kelpie can be leveraged in HPC codes is simply as a mechanism by which data can be shared between legacy applications. Most of today's HPC codes are massive and cannot easily be refactored to take advantage of new programming paradigms or communication facilities. However, many HPC users are interested in higher-level workflows that utilize a series of simulations to answer larger questions. Building a workflow out of existing simulations is difficult. While some researchers can merge their simulation codes into a monolithic executable that performs different functions as the workflow progresses, many codes are simply too difficult to fuse together due to library and namespace conflicts. As a result, a common approach to building workflows is to schedule different simulations to run at different times, routing intermediate data through the file system.

In-memory data stores represent an opportunity for improving these workflows, as the store can function as a convenient place for exchanging data between simulations in a high-performance manner. In the most basic case, an array of nodes can be dedicated to housing data outside of the individual simulation nodes. Simulation nodes can then interact with the array through simple client-based communications. A more advanced approach would

be to instead instrument the compute nodes with Kelpie server services, and store references to local data using shared pointers. This technique minimizes the amount of data that is transferred out of each node, and does not tax the resources of the compute nodes. In FY14 we considered two examples for workflow-based integration: assisting Reduced-Order Model (ROM) building and offloading data for in-situ analysis.

4.1.1 Reduced-Order Model Building

A number of SNL researchers are interested in utilizing *Reduced-Order Models (ROMs)* as a way to accelerate the exploration of large parameter spaces. At the most abstract level, a ROM is a computationally less-expensive proxy for a full-order computation. In Finite Elements, a ROM approximates the original solution space S with a subset of S. Building the ROM involves locating the important subdomains of the full-order problem and then discarding or simplifying at least some of the remaining subdomains. Procedurally, a ROM is typically built from a series of solutions to the full-scale $Ax = b$ problem with varying material and boundary conditions, where *(i)* A is a matrix that represents the physics of the system, *(ii)* b is a vector that represents the externally applied conditions, and *(iii)* x is a vector that represents the solution or "state vector".

The ROM-building process provides a good example of a workflow that can benefit from a distributed, in-memory data store like Kelpie. Current ROM-building data flows use file I/O to move data between different stages: typically, a full-order computation produces a number of large parallel output file that are then parsed by ROM generators. In most cases, only a small portion of the file data is used to generate each ROM (e.g., geometry and topology are not typically used). Thus, it is desirable to skip writing data to disk and have the ROM generators simply pull data out of intermediate memory as needed. Both Kevin Carlberg and Jeremy Templeton at SNL/CA have expressed an interest in how Kelpie could be utilized to house this intermediate data in their ROM-building workflows. In discussions with these researchers, we have noted that there are some difficulties that we would need to overcome:

Adapting to Online Workflows: Current ROM generation is typically performed offline, and therefore there has not been a driving need to optimize the ROM-building workflow. In order to prototype an online workflow, it would be worthwhile to revisit how ROM generation could be completed using existing libraries. Both Sierra and Trilinos have libraries that can perform the Tall-Skinny SVD operations this work requires.

The M→N Data Problem: For these users, the full-order simulation data is currently housed in EPetra/TPetra objects that distribute matrix and vector data across many nodes in the system. If the ROM generation work is not done in situ, it will be necessary to construct mechanisms that move data from the original M nodes into an array of N nodes during the ROM-building processes.

Designing Kelpie Data Interfaces: In order to store and extract data for a key/value

store it is necessary to devise a labeling system that makes it possible to store and retrieve components in an optimal manner. For instance in ROM building, we must be able to store the distributed state vector of the full-order model in Kelpie and be able to read specific sections of the data in a different collection of nodes. Both sides of this work must use the same labels and adhere to a decomposition that is suitable for both tasks. Additionally, it will be necessary to construct mechanisms that serialize the components and pass notifications when different regions become available.

An advantage of using a data store like Kelpie for building ROMs is that it allows us to consider *concurrent* ROM building. That is, given data form the first "i" full-order computations, one can compute a partial ROM which could reveal the more sensitive regions of the parameter study range. This information could be used to focus future full-order computations on the most sensitive inputs values, thus providing a more robust ROM and a more accurate one. This could also help reduce the burden on the analysts to know more about the system before creating a ROM building parameter study. In our discussions with local ROM builders, it is clear that this is unexplored territory and something that would be of interest to the community. It is our intention to explore this in more detail in FY15.

4.1.2 Online Data Analysis and Visualization

Another example of how Kelpie can be integrated into existing workflows can be found in scenarios where users want to inspect a simulation's data before execution completes. There are a number of approaches that HPC users employ today to instrument and analyze their simulations. A common approach is to work *in-situ* by instrumenting the actual compute nodes in a simulation with analysis codes. This work typically involves stripping a visualization package such as VTK down to minimal components and implementing analysis codes that can link in and coexist with simulation codes. Another approach is to work *in-transit* by implementing data processing algorithms as stream operators that are applied as results are written through I/O service nodes. We believe in-memory stores offer a middle ground to these approaches, as they can provide the ability to retrieve data from the running nodes, as well as buffer data and perform compute operations on other analysis nodes in the system.

This year we briefly looked into the logistics of how we could implement such a system on top of Kelpie, for the purpose of helping us plan out future capabilities of the library. We primarily focused on how existing mesh I/O libraries organize data as it is transitioned between file formats and visualization tools. VTK is well-utilized within SNL for mesh-based data processing and provides an example of how on-disk and in-memory formats differ. Mesh data files typically store related items together (e.g., multiple vectors of point or element data) while in-memory systems organize the data in ways that make it easy to render (e.g., organizing data so that it is easy to extract all of an element's variables). From this work we note that it is possible to use Kelpie's labeling mechanisms to restructure how data is organized in the store, and that we will need to define mechanisms for embedding complex objects into key/value entries.

4.2 Library-Based Integration

Another approach to integrating Kelpie into today's codes is through a library-based approach. Most HPC codes are built on top of complex libraries that implement common operations in a distributed manner. For example, EPetra/TPetra provide a way to store and manipulate data structures that are spread across many nodes in a distributed system. Constructing an alternate implementation of the library using Kelpie primitives therefore provides an easy way to try out Kelpie-based software with existing applications. While we expect that in most cases Kelpie will not meet the same performance levels that existing, well-tuned libraries offer, there are other reasons why substituting Kelpie in for data transfers can be valuable. First, it is possible that Kelpie's data distribution mechanisms can provide system benefits, such as improved resilience or better locality. Second, storing data in Kelpie may make the application better suited for tasks such as online data analysis. For our library-based integration work, we focused on working with Trilinos's EPetra/TPetra packages.

4.2.1 Trilinos: EPetra/TPetra

Many of SNL's applications are based on the Trilinos packages EPetra and TPetra (referenced here as E/TPetra), both of which provide support for distributed matrices and vectors. TPetra is a templated version of EPetra and will likely take over the bulk of use in new applications. From a developer's perspective, E/TPetra are appealing libraries to work with because they allow users to scale their codes up to parallel architectures without having to worry about all of the implementation details. In an effort to learn more about E/TPetra communications patterns, we wrote *EpetraKelpieCommunicator*, a drop-in replacement for *EpetraMPICommunicator*. This communicator allows us to move EPetra data between the distributed nodes of a single computation, or to provide a communication bridge to a completely separate application (e.g., as described in Section 4.1). In order to accomplish this work, we needed to work through the following steps.

Object Serialization: The first step in utilizing Kelpie to transport EPetra objects was to construct a serialization scheme for transporting and storing different items in Kelpie. For our examples we found EPetra data structures were not complicated and therefore data objects could be mapped directly to Kelpie key/value pairs. In the future it may be worthwhile to reconsider this choice and attempt to package multiple components together in the same key/value pair for efficiency.

Key Definition Strategy: The next challenge was to define a system for labeling different EPetra data components with Kelpie keys. This task has a number of trade-offs, as keys affect both how data is distributed in a Kelpie store, as well as how difficult it is for our communicator to fetch different components. We utilized 1D keys for this work and devised a labeling scheme that systematically identified each component of

data based on a combination of its physical and mathematical quantities, as well as its time-step information.

Communicator Interface: The final challenge was to construct the software interfaces that move data between EPetra and Kelpie. Fortunately, this task was relatively straightforward as EPetra provides a communicator interface where third parties can define their own means of moving data on EPetra's behalf.

We tested the EPetraKelpieCommunicator in a *Belos* (a Trilinos CG solver) solver example. This work demonstrated that the EPetraKelpieCommnicator provided correct data transfer across nodes for a Trilinos solver. Although Kelpie was much slower than MPI in performing the same task, we did learn about T/EPetra as well as useful Kelpie design patterns. Looking forward, we note that other Nessie developers are investigating how E/T-Petra communication can be improved. Specifically, Jay Lofstead is actively exploring how Nessie can be utilized to make TPetra data structures more accessible to other applications. We hope to leverage Jay's work on M-to-N problems with TPetra in future Kelpie work.

4.2.2 Influence on Other Packages

The work with EPetra was a positive experience for us and encouraged us to think about how library-based integration techniques can have a broad impact at SNL. The following are a list of E/TPetra applications that we have looked at that may offer other opportunities for Kelpie use.

Albany: Albany is an exploration of modern C++ techniques used in computational mechanics applications. Albany contains applications to model a variety of physical phenomena, many of which are non-linear. Albany is an open-source package. Of particular interest are the Albany applications used in ROM building research. Much of the ROM-building work by Kevin Carlberg described in Section 4.1.1 leverages Albany.

Mantevo: Mantevo [12] has become SNL's preferred testbed for exploring node computation models. Mantevo applications have simplified physics, topological structure, and geometric spatial domains. Simplifying these steps makes it easier to explore different models of computation. While Kelpie can easily be integrated into Mantevo through our EpetraKelpieCommunicator, we believe there are greater opportunities for leveraging Kelpie if custom communication options are explored.

Sundance: Sundance [14] provides another application from which EPetra/TPetra data can be extracted. Sundance provides the capability to solve a wide array of fluid and solid mechanics problems. We are targeting Sundance as a convenient tool to provide computations for Jeremy Templeton's ROM building procedure which was originally done using Sierra tools. Sundance provides an appealing surrogate for this work, as it is not encumbered with Sierra's licensing restrictions.

4.3 Custom Data-Plane Integration

The last technique we looked at for integrating in-memory data stores into HPC applications is to use Kelpie as a custom *data plane* within an application. At a fundamental level, distributed applications use communication libraries such as MPI for two types of operations: *(i)* to issue control directives that affect the flow of execution within the application, and *(ii)* to migrate data from one node to another. It can be beneficial to separate these functions and think of an application having both a *control plane* and a *data plane* within the communication fabric. Doing so provides an opportunity to define higher-level abstractions that make it easier for developers to implement their applications. For example, asynchronous task-DAG frameworks largely focus on the control plane, using it to dispatch work to different resources in the system. Similarly, data-intensive frameworks focus on using the data plane to ensure objects can be distributed and retrieved within the system.

Kelpie provides a rich communication fabric that allows users to develop their own custom data plane without having to micromanage all of its underlying network operations. While the complexity of rewriting legacy applications to take advantage of this capability is beyond the scope of this project, it is possible to explore how new applications could be developed to take advantage of Kelpie's native operations. This year we focused on two example scenarios. First, we prototyped how a few common, distributed solver operations could be implemented in Kelpie to evaluate whether our interfaces made it easier to develop and think about distributed computations. Second, we worked with the DHARMA team throughout the year and discussed different ways their asynchronous task-DAG work could leverage Kelpie. While this work is largely still in the planning stages, it has given insight into how we can open up Kelpie to make it more applicable to other developers.

4.3.1 Using Kelpie for Symbolic Work in Solvers

One of the advantages of utilizing key/value data stores to manage data in a distributed system is that they can enable developers to think about a problem at a higher level of abstraction. Rather than focus on the specifics of where data is located in a system, developers can simply reference the data and be assured that it will arrive when it becomes available. By using keys to label objects, work with a key/value store feels more like *symbolic programming*. While it is true that working with these higher-level abstractions can result in performance losses compared to ideal MPI implementations, there are many instances where developer productivity is much more important than peak runtime efficiency.

We explored multiple examples that looked at how we could use Kelpie to implement a custom data plane within an application. The examples are summarized as follows.

Simple Node-to-Node: Our first example simply demonstrated how Kelpie could push data from one node to another. This "HelloWorld" program serves as a reference for starting up the system and explaining how data can be passed between nodes with

blocking and non-blocking transfers.

Distributed Dot Product: This example performed a simple dot product computation on distributed vector data. Large vectors are divided into smaller regions that are identified by unique key labels. The dot product retrieves each region as needed until the final result is produced.

Distributed Matrix-Vector: This operation builds on the dot product above. However, the final vector has contributions from each node involved in the operation.

Distributed Matrix-Matrix: This is often used as a way to more efficiently compute the matrix-vector of a single matrix with a number of vectors arranged into columns of a matrix.

Distributed Conjugate Gradient: This example solves "Ax=b" for positive, definite, symmetric matrices, found in some basic Finite Element problems. Symmetry may be sacrificed for various constraints where other solvers should be employed.

The correctness of these operations demonstrated Kelpie's correctness in real-world scenarios. Developing these toy examples has provided us with insight into how complete codes could leverage an in-memory store. From this work we highlight two enhancements that we can make to improve Kelpie in the future. First, locality information is extremely useful for performance. We originally performed this work using a plain DHT (Distributed Hash Table) that distributed data across the nodes, but we did not leverage the locality information to steer our calculations. It would be more useful to locate where the data resides and then dispatch the calculation to the node. It would also be useful to construct other resource interfaces into Kelpie that allowed us to steer placement. Second, it would be useful to construct wrappers around Kelpie's resource interfaces that would implement higher-level data interfaces. For example, it would be beneficial to implement an interface that mapped a known API (e.g., Exodus or VTK) to Kelpie operations. These interfaces would likely require more flexible C++ mechanisms, such a functors or function objects, and with C++11, Lambdas, to handle Kelpie operations behind the scenes while presenting users with object-oriented data manipulation hooks. This is in line with how T/EPetra handle basic operations.

4.3.2 DHARMA: A Task-DAG Framework

The other custom data plane work we were involved with this year came from our interactions with the DHARMA team. In order to address reliability problems anticipated in next-generation computing platforms, DHARMA is developing a framework that will execute complex, directed, acyclic graphs (DAGs) of tasks. In his model, users will decompose their computations into a collection of tasks and then use DHARMA's runtime to efficiently dispatch the calculations to the resources that are available in the system. The advantage of this approach is that the system will be able to automatically reschedule portions of the

task DAG when components in the system fail, as opposed to resurrecting the entire job from the last known checkpoint. The work involved in developing the DHARMA framework is significant. Therefore, leveraging an existing data store like Kelpie as a way to maintain data reliably would be of great benefit to the project.

In FY14 Kelpie team members worked with the DHARMA team to help identify the technical features they required in order to build a running system. The DHARMA team focused on running system simulations that helped them explore the different ways their system could leverage a distribute in-memory store for maintaining data. One of the appealing results of this work is the notion of a *reference-based DHT*. In a traditional DHT, a node that wants to make data available to others typically pushes data to a node in the DHT that other nodes can reference. This approach spreads the load across the system but requires moving data objects through an intermediate node to facilitate a transfer from source to destination. The DHARMA team proposed a reference-based DHT as an alternative. In this DHT, the node generating the data keeps a copy of the data locally, and then publishes metadata about it to a node in the DHT. A reader simply queries the DHT to discover information, and then communicates with the node that generates the data to retrieve it.

After simulating the reference-based DHT, John Floren set about constructing a prototype of it in Kelpie. This work is still in progress, but it has provided valuable insight into how we must make it possible for developers to construct their own enhancements to the library. Our experiences with DHARMA have been extremely positive, as the task-DAG work is a natural candidate for key/value data stores. DHARMA uses its own labels to identify data, which are well suited for Kelpie's keys. DHARMA's concerns for performance have motivated us to consider new features (e.g., complex callback chains) and have helped us refocus our efforts to achieve better performance. As we move into FY15, we expect to work closely with DHARMA at building a system that will make their task-DAG work feasible.

References

[1] Apache accumulo. `https://accumulo.apache.org/`.

[2] Avro. `http://avro.apache.org/`.

[3] Cap'n proto. `http://kentonv.github.io/capnproto`.

[4] Cereal - a c++11 library for serialization. `http://uscilab.github.io/cereal`.

[5] Kyoto cabinet. `http://fallabs.com/kyotocabinet/`.

[6] Redis. `http://redis.io`.

[7] Aditya Agarwal, Mark Slee, and Marc Kwiatkowski. Thrift: Scalable cross-language services implementation. Technical report, Facebook, April 2007.

[8] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson Hsieh, Deborah Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[9] Kristina Chodorow. *MongoDB: the definitive guide*. "O'Reilly Media, Inc.", 2013.

[10] Jeffrey Dean and Sanjay Ghemawat. leveldb–a fast and lightweight key/value database library by google, 2011.

[11] Brad Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124):5, 2004.

[12] Michael Heroux, Douglas Doerfler, Paul Crozier, James Willenbring, Carter Edwards, Alan Williams, Mahesh Rajan, Eric Keiter, Heidi Thornquist, and Robert Numrich. Improving performance via mini-applications. *Sandia National Laboratories Technical Report SAND2009-5574*, 2009.

[13] Jay Lofstead, Ron Oldfield, Todd Kordenbrock, and Charles Reiss. Extending scalability of collective io through nessie and staging. In *Proceedings of the sixth workshop on Parallel Data Storage*, pages 7–12. ACM, 2011.

[14] Kevin Long. Sundance 2.0 tutorial. *Sandia National Laboratories Technical Report SAND2004-4793*, 2004.

[15] Sage Weil. *Ceph: Reliable, scalable, and high-performance distributed storage*. PhD thesis, University of California Santa Cruz, 2007.

[16] Sage Weil, Andrew Leung, Scott Brandt, and Carlos Maltzahn. Rados: a scalable, reliable storage service for petabyte-scale storage clusters. In *Proceedings of the 2nd international workshop on Petascale data storage: held in conjunction with Supercomputing'07*, pages 35–44. ACM, 2007.

DISTRIBUTION:

1	MS 1319	Jay Lofstead, 1423
1	MS 1327	Ron Oldfield, 1461
1	MS 9152	Robert Clay, 8953
1	MS 9152	Shyamali Mukherjee, 8953
1	MS 9152	Craig Ulmer, 8953
1	MS 9159	Gary Templet, 8954
1	MS 0899	Technical Library, 8944 (electronic copy)

This page intentionally left blank.

Sandia National Laboratories