SANDIA REPORT

SAND2021-1452
Unlimited Release
Printed September 2015

ATDM Data Management FY2015: Data Warehouse Progress Report

Nathan Fabian
Todd Kordenbrock
Shyamali Mukherjee
Ron Oldfield (PM)
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.

Approved for public release; further dissemination unlimited.

Sandia National Laboratories

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

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

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

SAND2021-1452
Unlimited Release
Printed September 2015

ATDM Data Management FY2015: Data Warehouse Progress Report

Nathan Fabian
Todd Kordenbrock
Shyamali Mukherjee
Ron Oldfield (PM)
Gary Templet
Craig Ulmer (PI)

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

Abstract

The Advanced Technology Development and Mitigation (ATDM) program at Sandia National Laboratories is a new effort to build next-generation simulation codes that will map well to upcoming exascale computing platforms. Rather than follow traditional single-program, multiple data (SPMD) programming techniques, ATDM is developing applications in an asynchronous many task (AMT) form that describes work as a graph of tasks that have data dependencies. The data management team is focused on developing a *data warehouse* for ATDM that will enable tasks to store and exchange data objects efficiently. This report summarizes the data management team's efforts during FY15, and documents: (1) an initial API and implementation for the data warehouse's key/value store, (2) API requirements for use with ATDM's runtime, (3) initial requirements for storing ATDM-specific data, and (4) the current organization of software components that will be used by the data warehouse.

Contents

1 Overview **7**

1.1 The Need for ATDM 7

1.2 ATDM Organization 8

1.3 Why Does ATDM Need a Data Warehouse? 10

1.4 Prior Data Warehouse Work 11

1.5 Data Warehouse Goals for FY15 11

2 Kelpie: Building a Key/Value Store for the Data Warehouse **13**

2.1 Kelpie Overview 13

2.1.1 Keys 14

2.1.2 LocalKV 14

2.1.3 Resources and Resource Handles 14

2.2 Kelpie API Use 16

2.2.1 Configuration and Initialization 16

2.2.2 Working with Resources 16

2.2.3 Obtaining a Resource Handle 19

2.2.4 Accessing Data 20

2.3 Kelpie Prototype 22

2.3.1 Resource Management 22

3 AMT Requirements for the Data Warehouse **23**

3.1 DHARMA AMT Overview 24

3.2 Data Warehouse Requirements 25

3.2.1 Data Labels 25

3.2.2 Memory Management 25

3.2.3 Underlying Communication Behavior 27

3.2.4 Data Distribution 27

3.2.5 Persistence 28

4 Application Requirements for the Data Warehouse 31

4.1 Application Interface Challenges 31

4.2 Application Use of the Data Warehouse 32

4.3 MeshDB Overview 32

4.4 PIC Overview 34

4.4.1 Particle Streaming 35

4.4.2 Mesh Split/Join 36

4.5 Data Warehouse Mechanics 37

4.6 Third-party Consumers and Producers 38

5 Data Warehouse Development 39

5.1 Identifying Areas for Improvement 39

5.2 Improving Data Warehouse Components 41

5.2.1 NNTI 3: A Portable RDMA Transport for HPC 42

5.2.2 NSSI Shim: A Thin RPC Layer for NNTI 43

5.2.3 Lunasa: Memory Allocator for RDMA Data 45

5.2.4 Webhook: A Reusable Debug/Control Interface 46

5.2.5 SBL: A Simplified Interface to Boost.Log 47

List of Figures

1.1 ATDM is organized into multiple areas of expertise, each with its own set of design teams. 8

2.1 Kelpie allows users to define multiple resource pools at the same time. 15

2.2 Users can easily change how Kelpie behaves by setting different key/value fields in the Configuration object used at initialization time. 17

2.3 Resource URLs provide information that can be used to reference and access a resource URL. 18

2.4 Resources can be set at start time by defining them inside of a Configuration passed in at initialization. 19

2.5 A ResourceManager encapsulates information necessary to communicate with a specific resource. 20

2.6 Users can issue blocking put and get operations through a resource handle. . 21

2.7 Users can perform asynchronous operations by issuing the request and then waiting on the result at a later point. 21

3.1 DHARMA manages all aspects of operation within the node and uses the data warehouse to orchestrate data transfers between nodes. 23

4.1 From a meshing perspective, an IOSS interface provides a standard way for both the MeshDB and applications to use the data warehouse. 33

5.1 Data warehouse software was decomposed into multiple components. 41

Chapter 1

Overview

The Advanced Technology Development and Mitigation (ATDM) program at Sandia National Laboratories is a new effort in ASC that is building next-generation simulation codes that will be able to take advantage of new capabilities found in upcoming exascale computing platforms. ATDM takes a different approach to parallel processing: rather than describe an application as a collection of distributed processes with explicit communication, ATDM asks users to organize their work into a graph of tasks with data dependencies that a runtime system can dynamically schedule on distributed resources. This asynchronous, many-task (AMT) approach presents an opportunity for a system to achieve better performance and reliability on exascale platforms, while reducing the difficulty associated with developing complex simulations.

The ATDM project covers the complete spectrum of what would be required to design, implement, and run ASC-relevant simulations using AMT techniques. This work includes simulation science (e.g., creating new codes to simulate and model complex systems in physical phenomena), modeling science (e.g., developing tools to represent the data in a way that simulations require), and computer science (e.g., designing the underlying software needed to run an AMT application efficiently on a distributed computing platform).

This document focuses on progress made in FY15 by the *data management* team in the Architectures and Software Development domain of ATDM. The data management team is responsible for developing a *data warehouse* that AMT applications can use to exchange their data objects. This effort addresses a number of low-level efficiency challenges, including migrating data between nodes, serializing data objects, managing memory, and defining APIs that balance performance with ease-of-use. Rather than start from scratch, the data management team focused on evaluating how prior work in ASC (e.g., Nessie and Kelpie) can be scaled up to meet ATDM's needs.

1.1 The Need for ATDM

ATDM was created out of concern that today's high-performance computing (HPC) programming techniques would not be effective on upcoming exascale computing platforms. Exascale systems will push the boundaries of computing and will likely offer a different hardware environment than what users see today. These systems will employ a massive number of compute nodes and rely on special-purpose hardware (e.g., GPUs and nonvolatile memory) to achieve their performance goals.

Today's HPC users largely develop single-program, multiple data (SPMD) applications that allow the user to explicitly define how computations and data flows take place in the system. While this fine-grained control enables users to micromanage how their programs execute, it can be challenging in complex applications to develop highly-efficient code that maximizes the resources that are available in a system. The sheer size and heterogeneity of upcoming systems compounds this problem and motivates us to consider approaches where an AMT system may do a better job of scheduling computations and orchestrating data flows than humans.

Resilience is another motivator for considering AMT solutions. Current codes are largely optimized to run in an error-free environment where a single node failure crashes the whole application and forces a restart from a previous checkpoint. Hardware errors are expected to be more noticeable in exascale platforms due to the increase in hardware of these systems. AMT approaches can help in these environments, as the runtime environment has more knowledge about how to rewind and redo individual tasks that have failed. While ATDM currently does not consider resilience to be the primary motivator for AMT, it is an extremely useful benefit that may become essential in future systems.

1.2 ATDM Organization

Making the transition to AMT requires a substantial research and development effort in a number of areas. Figure 1.1 presents a high-level view of different areas of work within ATDM.

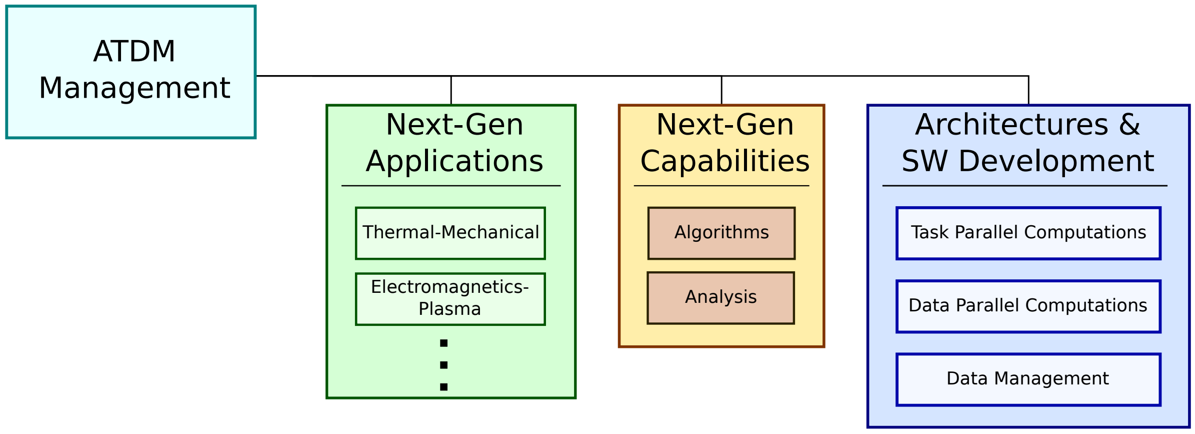


Figure 1.1. ATDM is organized into multiple areas of expertise, each with its own set of design teams.

At a high level, the different domains within ATDM are defined as follows:

Management: The management team oversees all work within ATDM and is responsible for interacting with external entities.

Next-Generation Applications: The next-generation applications effort is responsible for refactoring different ASC applications into AMT forms. Work this year largely focused on particle-in-cell (PIC) methods for electromagnetic simulations.

Next-Generation Capabilities: The next-generation capabilities effort is developing additional software that is necessary for supporting applications. For example, in the algorithms portion of this effort, a number of meshing experts this year architected a workflow that reads meshing specifications from existing libraries and populates a **MeshDB** that ATDM applications can use. This effort will interface with the data management team in FY16.

Architectures and Software Development: This effort is responsible for the majority of the *computer science* related work that is necessary for making ATDM realizable. The teams are:

1. **Task Parallel Computations (DHARMA):** DHARMA is responsible for defining all aspects of the AMT and constructing a runtime that will manage all of the other components in the system. In FY15 the DHARMA team focused on evaluating existing AMT systems and ported different Sandia mini apps to these systems to explore implementation trade-offs. This work has motivated the need for a more advanced AMT system that will be more appropriate for Sandia's ASC needs.
2. **Data Parallel Computations (Kokkos):** In order to maximize on-node computing performance, it is necessary to utilize tools that can efficiently map computations to underlying hardware accelerators such as GPUs or SIMD arrays. Kokkos [2] is an established effort at Sandia that has developed software that makes it easy to leverage data-parallel hardware, without being tied to one specific technology. Application developers will utilize Kokkos to make their individual tasks run as fast as possible.
3. **Data Management (Data Warehouse):** The data warehouse is responsible for managing all data objects that are produced and consumed by distributed tasks in the AMT runtime. The data warehouse provides the mechanisms by which DHARMA moves data objects between nodes in the system. It must be capable of supporting multiple, application-specific data interfaces in order to support the different data flows in ATDM applications.

This report focuses on the data management team's efforts. Given that the end product of this work will be a data warehouse for ATDM, data management is synonymous with data warehouse in our discussions.

1.3 Why Does ATDM Need a Data Warehouse?

For traditional parallel computing users, there is often some uncertainty about what is meant by the term *data warehouse.* The data management team is often asked the same questions:

What is a data warehouse? A data warehouse is simply a communication package that allows distributed applications to exchange data objects with each other in a reliable manner, without having to be specific about when or where those objects are generated or consumed. A data warehouse operates at a higher level of abstraction than explicit message passing: rather than require users to use directed communication in their applications (e.g., send *this message* to *that node* with *this tag*), a data warehouse distributes objects based on labels (e,g, make *this data object* available to any node that requests *this label*). This abstraction enables the communication library to do more on behalf of the user.

Is the data warehouse just for persistent data? No, the data warehouse is largely focused on managing *in-memory* data that a distributed application uses at runtime. It uses low-level communication mechanisms such as RDMAs to move data efficiently from one node's memory to another. However, persistence *is also* a natural operation for the data warehouse to manage. If a user tags an object as being persistent, the data warehouse can automatically replicate the object to either nonvolatile memory or the backing store of the platform's I/O subsystem.

Why does ATDM need a data warehouse? Unlike traditional message passing applications, ATDM decomposes an application into a number of tasks that consume and produce labeled data objects. A task has no knowledge about where its data objects came from or where they are going to next. This property provides a great deal of freedom for the AMT runtime to schedule how work and data transfers are performed in the system. At some level, the AMT needs a way to manage how data objects are moved about in the system.

While it is possible to implement both the scheduling operations and data transfer mechanisms needed by ATDM in the AMT runtime, it is valuable to separate the data object management functionality into its own entity. Distributed data object management rapidly grows in complexity as more production-related requirements are added: *How are race conditions resolved? How are objects deallocated? Can the system be hardened to support fault tolerance? How are nonvolatile memory resources and persistent storage leveraged?* Resolving these issues in a separate data warehouse component allows the AMT runtime to concentrate on higher-level problems, such as scheduling work and data flow on distributed resources.

1.4 Prior Data Warehouse Work

Members of our data management team have been working on data warehouse problems relating to HPC I/O for a number of years. There are two software development efforts in particular that are directly relevant to the construction of a new data warehouse for ATDM:

Nessie: Nessie [5] is a communication library that was designed to make it easy to write portable I/O services on top of different HPC platforms. Nessie is composed of two components: a low-level RDMA library named NNTI and a general purpose RPC library named NSSI. NNTI is a portability library for RDMA that includes drivers for a number of different network substrates (e.g., Blue Gene, Cray, and InfiniBand). While data services can be written entirely with NNTI, RDMA communication can be challenging to orchestrate due to the complexities of remote memory management. As such, Nessie's NSSI layer provides a more familiar RPC API that performs many useful operations for the user, such as automatic memory registration, fragmentation, and argument/result packing. Compared to other RPC libraries, NSSI is appealing because it still presents users with RDMA primitives for efficiently moving data in the system. As such, RPC writers have greater control over how large amounts of data are moved between nodes.

Kelpie: Kelpie is a distributed, in-memory object store that is enables users to move data objects between compute nodes in a flexible and efficient manner. Kelpie utilizes Nessie for its underlying communication operations and is therefore usable on today's HPC platforms. Data objects in Kelpie are referenced by a user-specified key, which is composed of three values: an application ID integer, a row ID string, and an optional column ID string. These fields allow users to organize their data and use the same store for multiple purposes. Kelpie provides built-in mechanisms for distributing data to different nodes (e.g., distributed hash table (DHT), broadcast group, reliable DHT), and can easily be extended with custom user-defined distribution policies.

1.5 Data Warehouse Goals for FY15

The first year of work in ATDM was largely focused on determining how the system should be constructed and developing prototypes for testing out how components will interact with each other. This report provides details on the following tasks that were defined for the data management team in FY15:

1. Design and document the application-programming interface for the key-value storage service. (Chapter 2)
2. Implement and demonstrate a simple prototype key-value storage service that uses available application memory to share data between tasks within an application. (Chapter 2)

3. Design and document abstractions/interfaces required to support task-based programming models and application resilience (e.g., local checkpoint/recovery). (Chapter 3)
4. Design and document abstractions/interfaces required for sharing mesh and particle data. (Chapter 4)

In preparation for work that will take place in FY16, Chapter 5 summarizes our effort to evaluate our current software and organize it into components that will be used to implement a data warehouse for ATDM applications.

Chapter 2

Kelpie: Building a Key/Value Store for the Data Warehouse

The distributed, in-memory object store (or key/value store) is the heart of the data warehouse and the component that received most of our attention in FY15. Given the momentum of our prior ASC I/O work, we decided to continue using Kelpie/Nessie as the basis for the in-memory object store. This chapter summarizes the design and implementation of a Kelpie prototype that was used for experiments during most of FY15. As the year progressed and more of the other ATDM components took shape, it was clear that there are changes that will need to be made to accommodate our users. Updates and a general discussion of the current status of the data warehouse are presented in Chapter 5.

2.1 Kelpie Overview

The core philosophy of Kelpie is to build a focused set of distributed memory management primitives that can be used to build more sophisticated data management services. The lower level of Kelpie embeds a variable-sized, in-memory key/value store in each rank of a parallel application in order to maintain data objects at each node, as directed by the application. A local key/value (or LocalKV) store can be manipulated locally through direct operations, or remotely through RPC/RDMA primitives. From a user's perspective, a data object stored in the LocalKV is simply a contiguous allocation of data that is labeled with a user-defined key. The lower level of Kelpie has no knowledge of where objects are located in the distributed system.

The higher level of Kelpie is responsible for how data objects are located in the distributed system. Kelpie allows a user to define multiple *resource pools* in the system. Each pool contains one or more nodes, and has a distribution policy specifying how objects should be mapped to resources. A common resource pool policy is to implement a *distributed hash table (DHT)* across a collection of nodes. When communicating with this DHT, the interface computes a hash of the object's key to determine the node where the object should reside. Users can extend Kelpie with their own functionality for controlling data distribution.

2.1.1 Keys

Kelpie utilizes a simple key structure to reference a user's data objects. Keys are composed of three fields: an application ID integer, a row ID string, and an optional column ID string. The application ID is a hash code that provides a namespace for data that Kelpie can use to isolate one set of data from another. In most cases an application will select an identifier at start time and use it for all of the application's work. This identifier helps prevent two concurrent applications from having naming collisions in the same Kelpie store. It can also be used as a crude access mechanism for controlling how data is shared between two concurrent applications: if an application wants to use data from another application, it must be given its application ID value. The application ID field is an optional parameter on most API calls and does not need to be explicitly set by the user.

The remaining two fields in the key are row and column identifiers in the store. These fields are variable-length[1] strings. The column portion of the key is optional and is provided as a simple way for users to group related items together. By themselves, keys only function as a way to label items, not dictate how they are distributed in the store. Distribution is controlled by the resource handles.

2.1.2 LocalKV

The LocalKV is a two-dimensional hash table that manages all of the bookkeeping for data objects that are stored at the local node. The application ID and the row ID are concatenated to index the first dimension of the table and retrieve information about all the columns in a particular row. The column ID of the key is used to select the corresponding column. This approach allows users to retrieve a specific row/column cell out of the table, as well as perform operations on all columns available for a particular row. If a request is made to retrieve an item that does not exist yet, the LocalKV can be directed to take later action (e.g., transmit to destinations) when the object does arrive.

In FY15 the LocalKV underwent significant modifications in order to take advantage of C++11 lambdas. The new implementation allows users to pass in a lambda operation to execute on a referenced object. This change simplified the LocalKV code and enables other parts of Kelpie to perform custom operations with the LocalKV.

2.1.3 Resources and Resource Handles

The upper layer of Kelpie is responsible for managing different resource pools that an application may utilize. Resource pools are simply a collection of one or more nodes that are responsible for storing a particular set of data objects in a way that is consistent to all nodes.

[1]Kelpie does have practical limits on how long these strings may be, due to the fact that large keys are time consuming to pack. Future versions will place lower bounds on length.

A resource pool is defined by a label, a distribution method, and the collection of nodes to use. Users access a data pool through a *resource handle*. A resource handle maintains configuration information about a pool (e.g., node IDs) and implements the means by which data is exchanged with the pool. All concrete resource handler classes implement a common set of communication functions and are responsible for interacting with the pool in the proper manner.

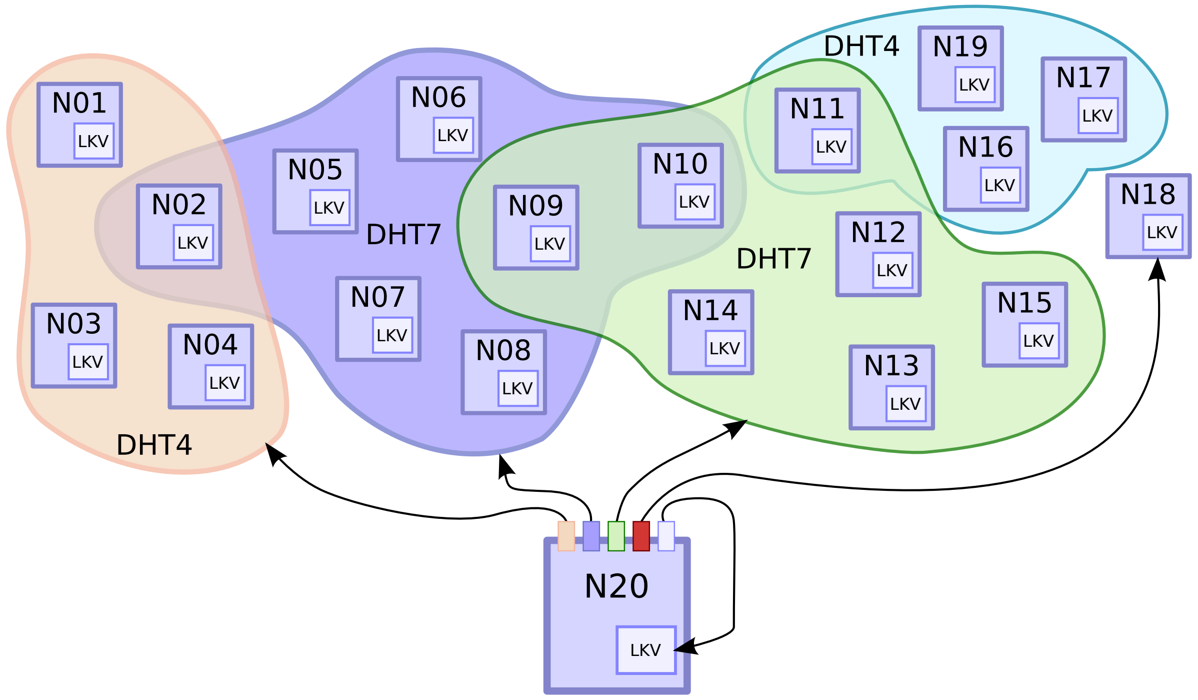


Figure 2.1. Kelpie allows users to define multiple resource pools at the same time.

Figure 2.1 illustrates how a node may interact with multiple resources pools that are distributed across different collections of nodes. In this scenario there are four DHT resource pools, some of which share common nodes. Node 20 has obtained resource handles to communicate with three of these DHTs, as well as a peer resource handle to talk to a specific node and a LocalKV handle to interact with its own store. When writing objects to the green DHT, the resource handle hashes the key of the object to determine which of the seven node will be the owner of the data. Given that resource pools are lightweight to construct, applications can easily establish small data communities for related items.

In FY15 we redesigned Kelpie's resource management (RM) service to allow better handling of dynamic resource registration. The RM now operates in a hierarchical manner and will automatically discover parent nodes in the resource path if a service is unknown. For example, if a resource is named "/a/b/c/d/mydht" and only the node "/a/b/c" is known, the resource management will consult "/a/b/c" to learn d, consult "/a/b/c/d" to learn mydht, and then interact with "/a/b/c/d/mydht" to get dynamic membership information for mydht.

2.2 Kelpie API Use

The full Kelpie API is available as a Doxygen-generated document that is part of the source code. This section provides a stripped down description of how users use the API to perform specific operations.

2.2.1 Configuration and Initialization

The `kelpie::Kelpie` class is the top-level class used to direct all Kelpie operations. Kelpie is a singleton (i.e., only one Kelpie instance is allowed at a time for a program or rank), as it holds specific network components that are tied to underlying hardware. A user typically creates the Kelpie object and then calls its `Init` function to dispatch the necessary services. Destroying the Kelpie object results in the termination of all network operations, which may affect other nodes in the system.

A Kelpie object is composed of a number of other components. It is often desirable to be able to customize how each of these components behaves in order to tune how a particular node operates (e.g., a server node may want to increase the default LocalKV memory limits to improve performance). Kelpie uses a `kelpie::Configuration` object to allow users to specify how different parameters are set in its components. The `Append` function allows users to pass in a multiple-line string of key/value parameters to set in the Configuration object. If a particular key is set multiple times, only the last value is utilized. For convenience, numerical values can utilize standard suffixes for large values (e.g., 1k = 1024).

Configuration uses a role-based approach to make it easier to define how different nodes in the system should behave. For example, in Figure 2.2 two roles are defined: a server and a client. The server sets the `rpc_server_type` parameter to single to specify that the node will be dedicated to RPC requests and will not need to spin the Kelpie RPC server off in its own thread. In contrast, clients are configured to not host any RPC services of their own. After the default configuration is programmed into the Configuration object, an additional `Append` operation is used to specify that node 0 will function as the server and all other nodes will function as clients. Configuration operations can be changed through appends until the Configuration is passed into a Kelpie initialization function.

2.2.2 Working with Resources

The goal of Kelpie is to make it easy for users to allocate one or more pools of distributed compute nodes, and utilize each pool as an application-specific *resource*. A resource in Kelpie is composed of three components: a resource name, a list of nodes that implement

```
// Multiple-line strings make it easy to define many things
string default_config = R"EOF(

  # Define the application id
  security_bucket               my_atdm_app

  # Server: run in a dedicated mode for hosting data
  # Client: do not allow others to connect
  server.rpc_server_type        single
  server.resource_manager.path  /myserver

  client.rpc_server_type        none
  client.resource_manager.path  /myserver/client1

)EOF";

main(){
  int mpi_rank;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);

  Configuration conf;
  conf.Append(default_config);
  conf.Append("node_role", (mpi_rank==0) ? "server" : "client");

  Kelpie kelpie;
  kelpie.Init(conf);
};
```

Figure 2.2. Users can easily change how Kelpie behaves by setting different key/value fields in the Configuration object used at initialization time.

the resource, and a client-side interface into the resource that defines how the resource behaves.

As illustrated in Figure 2.3, a resource URL may contain five pieces of information:

Resource Type : The resource type defines the client-side interface that is to be used to interact with the nodes. Kelpie currently provides three built-in handlers: lkv (for use with the node's LocalKV store), peer (for communication with a specific node), and dht (for communication with a distributed hash table of nodes). Users can define their own interfaces and register them with Kelpie as a new Resource Type.

Node ID: A URL may include one node ID to identify the node that is responsible for a particular resource. When the Node ID field is not present in a URL, Kelpie must discover the location of the node by walking the resource's path and consulting with

Resource Type:	<Node ID>	[Application ID]	/Resource/Path	Options

Examples:

lkv:	Use LocalKV with default settings
lkv:[myappid]	Use LocalKV, but set Application ID to myappid
peer:<0x0123>[myappid2]/a/b	Communicate with a specific host
dht:/a/b/mydht	Interact with a DHT (discovering details first)
dht:<0x4567>/a/mydht&nodes=4	Specific DHT information

Figure 2.3. Resource URLs provide information that can be used to reference and access a resource URL.

ancestors.

Application ID: A specific application ID can be defined in the URL to help in instances where data is moved from one application's namespace to another's. String values are automatically hashed to the numeric application ID value. The application's default value is used if this field is left unspecified.

Resource Path: The Resource Path is a "/"-delimited string that allows users to organize their resources into a tree-based hierarchy.

Options: The options portion of the URL provides a means for users to encode different configuration operations into a single URL. Options are separated by an ampersand and may be any string value of the user's choosing. Options can simplify query/response services, as a user may send a sparse URL (e.g., "dht:/a/b/mydht") to a node to find more information, and be returned a URL with more details (e.g., "dht:[myappid]<0x1234>/a/b/mydht?num_nodes=4").

Kelpie provides two means by which users can define, locate, and use resources. First, a user can define resources in a static way when Kelpie initializes, as illustrated in Figure 2.4. This approach is used for small installations and debugging, and simply involves users defining one or more resource URLs in a Configuration passed to Kelpie. These resources are shared through simple file operations and are only intended to simplify the bootstrapping required to get a small environment running.

The preferred way to manage resources is for applications to dynamically manage them as needed. In this approach one or more root nodes are defined for each application (e.g., "/a", "/b", etc.) and made known to all relevant nodes. Resources can then be added to the path in a hierarchical manner as needed. Because Kelpie will automatically walk the resource path to discover unknown resources, users simply request a resource to locate it. Nodes can request to join, leave, or get information about a particular resource.

```
// Multiple-Line strings make it easy to define many things
string default_config = R"EOF(

  # Server is the root in the tree and hosts a dht
  server.resource_manager.hosting        /a
  server.resource_manager.hosting        dht:/a/mydht?num_nodes=2
  server.resource_manager.write_to_file  .server-id

  # Client reads config info from a file
  client.resource_manager.read_from_file .server-id

)EOF";

main(){
  int mpi_rank;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);

  Configuration conf;
  conf.Append(default_config);
  conf.Append("node_role", (mpi_rank==0) ? "server" : "client");

  Kelpie kelpie;
  kelpie.Init(conf);

  if(mpi_rank==1){
    ResourceManager *rm;
    kelpie.GetComponents(NULL, NULL, &rm);
    rm->Join("dht:/a/mydht");
    // ...
    rm->Leave("dht:/a/mydht");
  }

  // ...

};
```

Figure 2.4. Resources can be set at start time by defining them inside of a Configuration passed in at initialization.

API Note: Future versions of Kelpie will migrate resource manager functionality to the Kelpie object for simplicity.

2.2.3 Obtaining a Resource Handle

A resource handle is the primary means by which a Kelpie user interacts with a resource. The `kelpie::Resource` class is the base class for all resources, and provides standard `put` and `get` semantics for retrieving data. As illustrated in Figure 2.5, a user can easily request a new resource handle simply by issuing a connect operation to Kelpie.

```
//..
Kelpie kelpie;
kelpie.Init(config);
Resource *dht = kelpie.Connect("dht:/my/dht");
//..
```

Figure 2.5. A ResourceManager encapsulates information necessary to communicate with a specific resource.

2.2.4 Accessing Data

Once a resource handle is obtained, a user can read or write data objects as needed. While Kelpie utilizes asynchronous operations for all communication, synchronous (i.e., blocking) API calls are provided for operations as well. Asynchronous operations require the user to maintain a `kelpie::RequestHandle` data structure. This structure maintains internal state information about a network operation and cannot be deallocated until the operation completes. The RequestHandle can be queried to retrieve information about the network operation (e.g., whether it was successful, whether the remote node fulfilled the request, etc.). A basic put/get example is listed in Figure 2.6.

Better overlap can be achieved by using Kelpie's asynchronous interfaces. As demonstrated in Figure 2.7, these operations require a user to issue one or more commands and then block on their completion.

API Note: Given the operation completion requirements defined by DHARMA, this interface will change in the near future. We expect to switch to supporting callbacks (possibly lambdas) for completion notification. Request handles may be removed entirely if callbacks present a cleaner interface.

```
rc_t rc;
Resource *dht = Kelpie.Connect("dht:/my/dht");

//Push out an object and wait until the transaction completes
rc = dht->PutSync(Key("myobj1"), &my_data, my_data_size);

//Make sure operation completed without errors
assert(rc==KELPIE_OK);

//Retrieve an object, blocking until it is available
RequestHandle req;
char myobj2[1024];
rc = dht->GetSync(Key("myobj2"), myobj, 1024, &req);

//Make sure operation completed without errors
assert(rc==KELPIE_OK);

//Get more info about the operation
cout << req.remote_rc          << endl
     << req.get.returned_bytes << endl
     << req.get.origin         << endl;
```

Figure 2.6. Users can issue blocking put and get operations through a resource handle.

```
Resource *dht = Kelpie.Connect("dht:/my/dht");

RequestHandle reqs[100];
for(int i=0; i<100; i++){
  dht->Put(Key(item_names[i]),
           data_ptrs[i], data_lens[i],
           &reqs[i]);
}
for(int i=0; i<100; i++){
  regs[i].Wait();
  //.. check return codes and results
}
```

Figure 2.7. Users can perform asynchronous operations by issuing the request and then waiting on the result at a later point.

2.3 Kelpie Prototype

The Kelpie implementation from FY14 served as an initial prototype for our data warehouse work throughout most of FY15. Given that this prototype was functional and the requirements for other ATDM components did not stabilize until the end of the year, we largely focused on enhancing Kelpie's resource management functionality. In this section we briefly summarize the work that went into improving the resource manager that was described previously.

2.3.1 Resource Management

The most challenging aspect of the Kelpie prototype this year was renovating its resource management facilities. The FY14 implementation's ResourceManager suffered from two main deficiencies:

Limited Attributes: The original resource manager only needed to maintain a simple list of nodes that belonged to a resource pool. While this approach was sufficient for basic resources, we realized that additional attributes needed to be associated with a resource and that applications would need better control or the memberships. We realized that URLs provided an excellent means by which these attributes could be modified and retrieved.

Implementing Hierarchy: The FY14 also lacked the ability for Kelpie to walk through a resource hierarchy and discover information that was not available locally. The system could query a remote node, but it could not work its way upwards to find gaps in knowledge.

The resource management portion of Kelpie was rewritten in order to fix these deficiencies. The changes utilize improvements to URL notation, and resulted in new caching structures that track information about different resources in the system.

Chapter 3

AMT Requirements for the Data Warehouse

The core strategy of ATDM is to transition ASC applications from a traditional single program, multiple data (SPMD) form into an asynchronous many-task (AMT) form that allows a runtime to manage how computations and data transfers are scheduled on parallel resources. The DHARMA team leads this work and is responsible for assessing the viability of existing AMT runtime systems and, if needed, developing a new AMT that would be appropriate for ATDM's applications on exascale platforms.

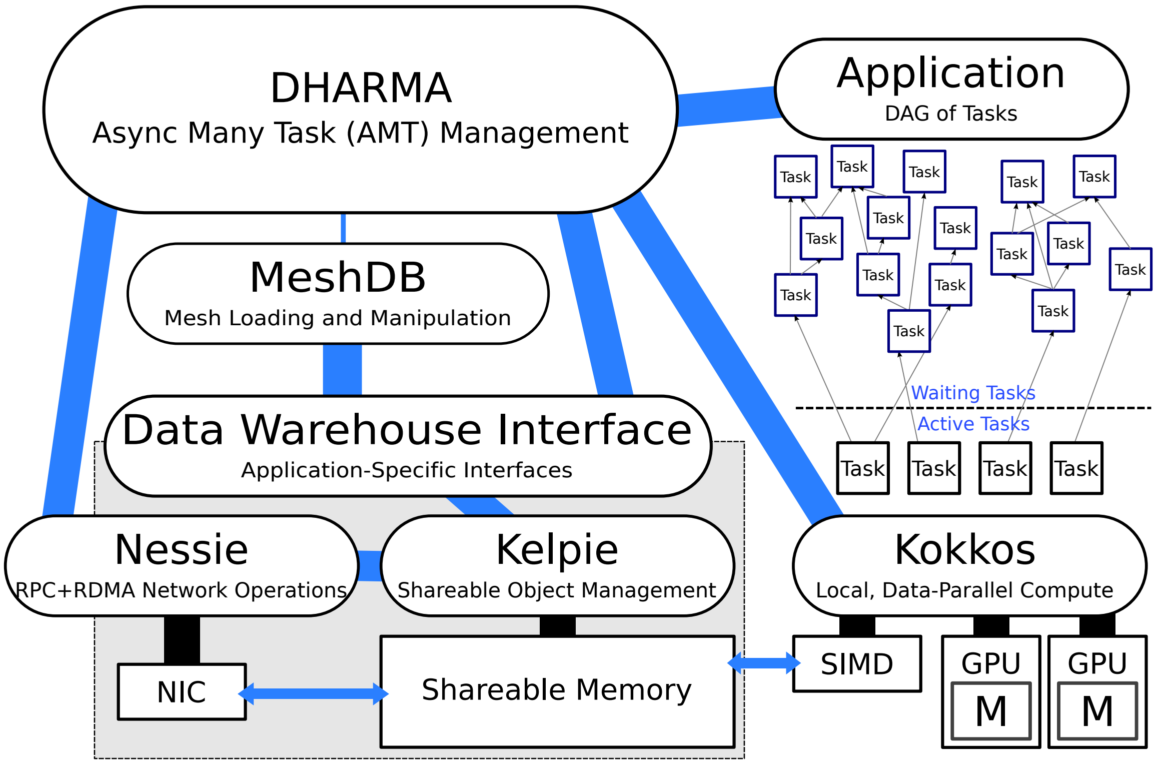

DHARMA
Async Many Task (AMT) Management
Application
DAG of Tasks
Task
MeshDB
Mesh Loading and Manipulation
Waiting Tasks
Active Tasks
Data Warehouse Interface
Application-Specific Interfaces
Nessie
RPC+RDMA Network Operations
Kelpie
Shareable Object Management
Kokkos
Local, Data-Parallel Compute
NIC
Shareable Memory
SIMD
GPU
M

Figure 3.1. DHARMA manages all aspects of operation within the node and uses the data warehouse to orchestrate data transfers between nodes.

As illustrated in Figure 3.1, DHARMA's AMT will manage all of the software components that are used to process ATDM applications. In the AMT approach, an application is defined as a collection of tasks that have data dependencies. DHARMA uses the data warehouse to resolve these dependencies so that it can schedule tasks on local data-parallel resources.

3.1 DHARMA AMT Overview

The DHARMA AMT runtime is responsible for managing all aspects of how an ATDM application is processed in a distributed system. Rather than define an application as a traditional single program, multiple data (SPMD) executable that explicitly passes data between different ranks in the simulation, DHARMA's AMT requires users to express an application as a collection of tasks that have data dependencies. These dependencies result in organizing the tasks into a directed acyclic graph (or task DAG) that the DHARMA runtime can use to schedule how work and data transfers take place in the distributed system.

A task is the fundamental unit of computing in DHARMA. Each task consumes zero or more input data objects and produces one or more output data object. An application-defined label (or key) is associated with a data object to allow the system to manage the application's data in a symbolic manner. In order to make the runtime more practical, tasks are idempotent: they do not modify their input data objects and they always produce the same outputs for a given set of inputs. Ideally, all possible input dependencies are defined for a task when it is created. Doing so enables DHARMA to defer the execution of a task until all of its inputs are available on the local node. However, DHARMA does permit running tasks to request data objects that were not in the original input dependency list. This feature is necessary in workloads where a variable amount of data is generated during execution, or in situations where a task inspects its inputs to locate additional pieces of data that are necessary for completing a task.

While it is expected that developers will take advantage of libraries such as Kokkos to leverage local, data-parallel hardware, a task only executes on a single node (i.e., a task is not a collection of MPI ranks that work in parallel). Ideally, all communication between tasks would take place as data warehouse operations, thereby allowing DHARMA to control all aspects of data flow through the system. From a practical perspective though, it is expected that this bookkeeping would be excessive and impede the performance of the system. As such, DHARMA will likely allow tasks to perform some form of lightweight communication. While the data warehouse's underlying communication layer may be used for these transfers, the data warehouse will not need to directly support these operations.

3.2 Data Warehouse Requirements

Based on the current design plans for DHARMA, it is useful to define core requirements for how the task DAG will utilize the data warehouse. The remainder of this chapter focuses on defining these requirements.

3.2.1 Data Labels

DHARMA uses application-defined labels to reference data objects that are used by a simulation. These labels have the following properties:

Variable-Length Keys: Data labels used by applications will be converted to a variable-length field that is 56 bytes or smaller. This length was chosen because it gives a large key space to work with, while at the same time being compact enough to fit in a cache line.

Multi-Dimensional Key Components: The key will contain multiple data fields that inherently make it multidimensional in nature. The first field is mandatory and is a variable name for the object.

Optional Fields: A key may be encoded to have optional fields, such as a version number. These fields may take on specific meaning for key/value operations (e.g., only maintain the latest two versions of an object).

Low-Dimensionality in Organization: While keys may have several dimensions, it is expected that the underlying key/value store will largely organize them in a two-dimensional manner (e.g., row/column). A mapping will be defined at a later point to determine how particular keys are grouped in the underlying storage.

External User ID: The key will not contain a user ID (or bucket) field to separate one application's data from another. When needed, this field will be managed at the client interface into the data warehouse.

3.2.2 Memory Management

The fundamental purpose of the data warehouse is to store data objects in a way that allows the objects to be retrieved in an efficient manner. RDMA transfers are desirable for facilitating quick remote access, but require users to store data structures in memory *registered* with the communication library. As such, the data warehouse must manage how data objects are housed in NIC-accessible memory. A functional system requires the data warehouse perform the following memory management functions:

Allocate/Deallocate NIC-Accessible Memory: The data warehouse must be able to allocate/deallocate variable amounts of memory that the NIC can access through RDMA methods.

Common Data Structure for Referencing Memory: The data warehouse needs to use a single data structure for referencing memory in the system. This structure must be capable of identifying both *local* and *remote* memory, although the structure *is not* responsible for resolving remote references by itself. Remote references will be resolved by the data warehouse and DHARMA.

Referencing Remote Memory: When used to reference remote memory, the memory data structure must:

1. Contain RDMA pointers and block lengths that can be used by the messaging layer to move data.
2. Include serialization mechanisms that allow the a memory reference to be easily packed in a message.
3. *Expected:* Include additional, application-specific information, such as an *expiration date* for the memory region.

Referencing Local Memory: When used to reference local memory, the memory data structure must:

1. Utilize some form of reference counting to ensure an object is not deallocated while in use.
2. Provide raw pointer semantics that allow a user to easily reference the data by virtual address.

Network Agnostic: All memory management must be done in a way that is network agnostic. Developers should not need to be aware of the specifics about the network transports they are using when referencing memory.

Expeditious: Allocation and deallocation need to be quick. Given that allocating network accessible memory can be time consuming, it is necessary for the system to preallocate memory blocks when possible.

No Enforcement of Idempotency: DHARMA is built on idempotent policies and has the expectation that applications will not modify previously generated data values. This policy *could* be enforced by the data warehouse by returning users with copies of data instead of references. However, this approach is expensive to the common case and may impede in optimizations where referenced memory can safely be modified. Therefore the system *will not* prevent users from modifying data handed back through references.

Directed Garbage Collection: DHARMA has detailed knowledge about how useful different data objects are in the system. As such, the data warehouse's memory management system will take guidance from DHARMA as to when objects can be deallocated or moved to persistent storage.

3.2.3 Underlying Communication Behavior

The data warehouse is responsible for migrating data between nodes as dictated by DHARMA and its applications. As such, it is important that the data warehouse has the following communication properties:

Asynchronous Operations: The data warehouse must utilize asynchronous communication primitives that allow multiple requests to be in flight at the same time.

Internal Bookkeeping: All information used to track outstanding messages must be maintained *within* the data warehouse. DHARMA will not keep lists of in-flight messages.

Callback-Driven Completion: In order to make communication more manageable, DHARMA will issue communication operations that include the communication action to perform (e.g., put or get) and the callback function to invoke when the action completes. These callbacks are currently expected to be traditional functions, although C++11 lambdas may be an option if additional flexibility is needed.

Expected Data Movement Operations: At the top level, the data warehouse will be responsible for moving data objects between nodes. Internally, there is great benefit in handling these operations in different ways. The top-level data movements include:

1. `Prefetch`: This operation notifies other nodes that this node should be forwarded a copy of the data when it becomes available.
2. `Publish`: This operation pushes an object to zero or more destinations, depending on the configuration and specifications of the client interface.
3. `Get`: This operation is used by DHARMA or an application to retrieve a local reference to a data object. If the object is not available, the data warehouse will retrieve it from a remote node.

While prefetch is a simplified version of get, it is worth distinguishing between the two as prefetch may use one-sided communications that incur less overhead.

It is expected that additional operations will be added to this list for more sophisticated data transfers. It is feasible to add operations that factor in key dimensionality (e.g., `GetRow`) or do data-specific manipulations (retrieve a portion of a row). These optimizations will be developed later as requested by DHARMA.

3.2.4 Data Distribution

There are multiple ways in which DHARMA may need to distribute data at runtime. Based on Kelpie's previous work, data distribution can be handled through custom, client-side interfaces that can be adapted to different situations. These interfaces have the following properties:

Uniform, Client-Side Interface: A client communication interface will implement all features needed to talk to a set of distributed resources. It will maintain the list of nodes participating in the resource as well as the underlying functions for implementing a particular communication strategy. Client communication interfaces will implement the `Prefetch`, `Publish`, and `Get` API described in the previous section.

Multiple Client Communication Interfaces: A task may instantiate multiple client communication interfaces at the same time to handle data transfers with different resources in the system.

Extensible: There are multiple ways in which DHARMA may need to manage collections of nodes for data distribution. It is therefore vital that users be able to define new client communication interfaces that implement different data transfer mechanisms. Examples of different client interfaces may include:

1. *Pure DHT:* A traditional DHT hashes a key to determine which end node is responsible for hosting a particular piece of data. While easy to implement, a pure DHT is inefficient for large objects, as an intermediate node must be used for communication between two nodes.
2. *Meta DHT:* In a Meta DHT, objects remain on the nodes that generate them, while metadata is published to DHT nodes. A client then consults the Meta DHT to locate information about where a particular object is hosted, and then performs an RDMA get to retieve the actual data.
3. *Reliable DHT:* A DHT can be made more resilient by replicating its data objects on different nodes. The simplest approach is to have clients push data to the node the key hashes to, and then push a copy to the next node in the collection.
4. *DHT with Backing Store:* A DHT can be extended to support persistent storage in a variety of ways. This operation may hash the operation to a particular node in the collection, and then delegate the job of storing data to disk to that node.

It is expected that the majority of this functionality can be implemented using a simple API. However, future work may need to expand the capabilities of the client communication interface.

3.2.5 Persistence

While DHARMA's immediate need for the data warehouse is in the role of in-memory data object storage, there is interest in extending it to be able to utilize both nonvolatile memory and parallel filesystem storage.

Burst Buffers: Emerging HPC platforms such as Trinity make large pools of flash memory storage available for intermediate storage. These resources present an opportunity for DHARMA to improve both memory utilization and resilience by offloading older data objects to the burst buffer. As such, the data warehouse should provide a mechanism for housing objects in the burst buffer.

Persistent Data Stores: DHARMA will need to be able to store important data objects off to persistent storage in order to save results and allow applications to be restarted. The data warehouse will need to be able to:

1. Store individual data objects as directed by DHARMA. This approach differs from the per-rank based approach favored in today's checkpoint/restart libraries.
2. Store metadata about data objects in a way that allows the data to be reloaded or inspected by other applications.
3. Take advantage of any built-in, multi-dimensional object store capabilities in the target I/O system (e.g., Ceph's RADOS) in order to allow better organization of data objects.

This page intentionally left blank.

Chapter 4

Application Requirements for the Data Warehouse

In addition to meeting the data movement requirements of DHARMA's AMT runtime, the data warehouse must also be capable of storing and retrieving *ATDM-specific data* in a way that is easy for our application developers to use. An examination of ATDM's applications and data access patterns reveals there are a number of challenges to building useful application interfaces into the data warehouse, and that there should be no expectation that a single API will serve all purposes. As such, we advocate constructing a small number of data-specific interfaces for the data warehouse that (1) provide the user with familiar APIs and (2) handle data distribution in a way that is useful to DHARMA. For this work we focus on the design of data interfaces and conceptual algorithms in support of two ATDM applications: a meshing database (or MeshDB) API for managing geometries and field data, and a particle data interface for use with particle-in-cell (PIC) simulations.

4.1 Application Interface Challenges

In prior task-DAG efforts, researchers have built custom data warehouses that are optimized for particular dataset types (e.g., unstructured grids). These systems work well for their intended applications because the data warehouse's interfaces are tightly coupled to the applications' needs. Unfortunately, these interfaces also make it difficult to use the task DAG software for applications that have different data needs. An examination of ATDM's applications reveals there are a number of challenges that will need to be addressed to make the data warehouse a success:

Multiple Types of Datasets: Different ATDM applications process different types of data. While many applications operate on mesh datasets, others manipulate their own types of data, such as particles. As such, the data warehouse cannot be optimized to handle just one type of dataset. It must be flexible enough to be customized to each application's needs.

Different Data Access Patterns: Even when applications use the same type of data, they may access the information in vastly different ways. Optimizing how data is distributed may help one application but penalize another. Therefore a single data interface may need to be customized to different runtime conditions.

Mixed Workloads: Some applications will need to access different types of data at the same time. For example, a PIC application may need to retrieve mesh data *and* exchange particle data among tasks. A data warehouse must be able to support both types of operation at the same time.

Users Expect Familiar APIs: ATDM's developers have a great deal of experience working with existing APIs and expect the data warehouse to have a familiar interface they can easily use. There is little overlap between some user communities, and therefore the data warehouse will need a way to support a small number of custom interfaces.

4.2 Application Use of the Data Warehouse

The challenges associated with ATDM's applications make it clear that it is unlikely that a single API will be sufficient for all data warehouse users. Instead, we focus on the development of a small number of *data interface modules* that will enable us to customize how different users leverage the data warehouse. Each data interface module is responsible for two functions: (1) providing the top-level API that end users see when they work with their data and (2) implementing the internal mechanisms by which data is decomposed into objects that can be managed and distributed by the data warehouse.

Figure 4.1 illustrates an example of how an application would use multiple data interface modules to interact with the data warehouse. In this example, the MeshDB team reads input data from external files, refines it, and then pushes the data to the data warehouse through an IOSS data interface module. This module allows the MeshDB team to use IOSS for their API and enables data warehouse developers to decompose mesh data into a collection of data objects that are distributed to different in-memory resources in the data warehouse. An application's tasks use two modules: an IOSS data interface module to read relevant sections of the mesh data and another application-specific data interface module to push results back into the data warehouse. A separate analysis application can then use the same application-specific module to retrieve the results as needed.

4.3 MeshDB Overview

The MeshDB team is using IOSS as a general interface into the data produced by the meshing tools. Following their lead, the data warehouse will also provide IOSS as an interface to ATDM's application components. This is an obvious decision as it reuses an interface familiar to ATDM developers and does not require the development and learning curve of a custom interface. We expect IOSS and the data warehouse to adapt as ATDM requirements

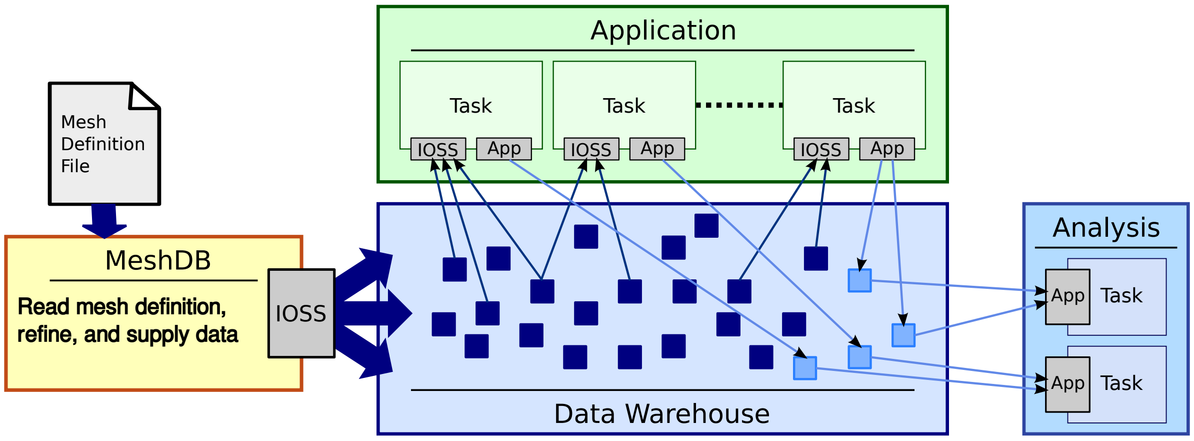


Figure 4.1. From a meshing perspective, an IOSS interface provides a standard way for both the MeshDB and applications to use the data warehouse.

develop. Several important points about the design of this module emerged during our discussions with the MeshDB team:

IOSS Data Warehouse *Back-End*: To allow ATDM components to read and write mesh data, fields, and particles, the data warehouse team will develop an IOSS *back-end*, which functions as a data translation layer between the data warehouse and ATDM components. This module is critical as it makes component data available for storage and communication by Kelpie.

Storage and Communication: In its current form IOSS does not support data warehouse storage and communication APIs. To provide this support we are investigating two options: (1) extend the IOSS interface to include storage and communication semantics, or (2) create a separate *data interface module* for storage and communication for use alongside IOSS. The latter option is more viable in the near term as it is less disruptive to other IOSS users. Interfaces to parallel meshing have been an active research area for some time. In that sense, ATDM requirements are unique as it seeks to reduce dependence on file I/O, and also introduces a parallel key/value store. Development of this interface can produce original work in this area.

Not a Parallel Mesh Object: In storing the mesh, the data warehouse becomes, in a very real sense, an *in-memory* parallel mesh object. It is tempting to seek a *do-it-all* data warehouse interface that offers a wide range of meshing and computational operations. However, this option is nearly impossible given the wide variety of mesh types (structured, unstructured, hybrid, etc.) and the diversity of computational operators needed by ATDM. Our goal instead is to focus on the immediate requirements for ATDM, and concentrate on data operations that reduce the burden of the component developer.

Mesh Resilience: The meshing interface has mixed resilience requirements due to the nature of its data. In general, the geometries generated by the MeshDB *do not* need to be made resilient to failure because the MeshDB should always be able to reconstruct the data based on external input files and application parameters. However, any field data that a simulation generates during its lifetime *may benefit* from redundancy to protect against failures. Decisions about when to replicate these objects will originate from DHARMA.

Persistence: Mesh data objects may need to be made persistent to (1) support resilience and (2) enable different components in a workflow to share information over extended periods of time. Mesh data objects must be stored in a way that is relevant to end users: some readers need a complete set of data fields for a particular timestep while others need multiple timesteps for a particular region of the mesh. Similar to *resilience*, persistence will need direction from a DHARMA-based workflow to determine which objects are made persistent.

There are a few points to consider in the context of the above solution strategy concepts:

- All of the solution strategies above are in their conceptual phases of development. We expect these ideas to grow and change through close collaboration with other ATDM component developers.
- Many of the data operations above will be controlled via the AMT either by having the data warehouse create AMT nodes, or the AMT tasks exercising the data warehouse interface.
- The development path should be to solve the data mechanics problem first and then decide what is the most natural ATDM component(s) to house it.

4.4 PIC Overview

ATDM is using particle-in-cell (PIC) methods to simulate plasma physics. PIC is a classic computational mechanics tool dating back to early *Hydrodynamic* codes developed at Los

Alamos [3]. PIC methods are based on the idea of decoupling the interactions of charged particles and E-M fields of a plasma. The particles encode a location in the computational domain and many other physical properties such as *charge* and *velocity*. The mesh provides a domain for computing and interpolating the E-M fields. While the particles move, the mesh is typically not moving, so decoupling the two is an intuitive approximation. PIC methods are generally broken into four computational steps:

Particle mover: Compute the motion of particles acted upon by the E-M field described on the mesh.

Charges and currents: Compute the charges and currents produced by the motion of the charged particles. These are interpolated to mesh topologies, nodes, edges, etc.

E-M fields: Compute the E-M field induced by the charges and currents associated with the mesh.

Field forces acting on particles: Compute the forces acting on the particles in the system, thus closing the system of equations and allowing this series of computation to repeat until equilibrium.

While these computational stages are sufficient for implementing a basic PIC simulation, there are additional advanced modeling capabilities that modern PIC codes support that make the code's implementation more complex. For example, a PIC code may include support for modeling *particle interactions*, which results in additional particles being generated and tracked during the lifetime of a simulation. Also, PIC codes may support the use of meshes of differing types (e.g., hybrid structured-unstructured meshes), which complicates how particles are applied to the meshes.

In terms of the data warehouse, the key hardship for handling data in PIC simulations is dealing with how particles are exchanged between geometric partitions of the domain. The number of particles exchanged and the frequency at which they are exchanged can vary wildly throughout the simulation. As such, using a data warehouse to route PIC data means our implementation must be able to (1) deal with variable amounts of data flow, and (2) perform *load balancing* to minimize communication costs when possible. Based on our interactions with the DHARMA and PIC application teams, we have outlined strategies for addressing two challenges associated with handling data flow for the PIC application: particle streaming and mesh splitting/joining.

4.4.1 Particle Streaming

In the ATDM PIC application, neighboring mesh partitions exchange a variable amount of particle data every time step. The problem with orchestrating these data exchanges is that a partition does not know how much data it will receive in a timestep or when it will arrive. A sender may need to push *multiple bundles* of particles to a neighbor in a single timestep (e.g., when fast-moving particles arrive from a neighbor after the partition's initial set of particles has already been transmitted). As such, a PIC data interface module cannot simply declare a static list of objects a task will need at the beginning of its timestep. Instead, it is necessary to devise a way of streaming variable amounts of data through the data warehouse.

After discussing the problem with the PIC application team, we believe one possible solution is to embed a *key management* system into the data interface module that can take advantage of Kelpie's 2D key notation. In this system the application interface would allow a task to query the data warehouse to locate any new bundles of particles that the task would need at a particular timestep. Internally, the data interface module would encode the destination and timestep values into the row portion of the key, and the source and microtimestep information into the column portion of the key. This naming avoids collisions by uniquely labeling each bundle, and enables Kelpie to do the work of grouping related items together in the same row. The system becomes even more efficient if the key/value store allows a receiver to subscribe to all activity in a particular row.

This type of streaming can be used for multiple forms of particle exchange. In addition to streaming particles between partitions, PIC applications may need to eject dead particles for efficiency and gather live particles for statistics. Application-specific details depend on how DHARMA's APIs evolve, but this strategy provides a basis upon which more complex schemes for communication can be built for PIC particles. We expect that there are many other data flows in ATDM applications where this streaming strategy will also be useful.

4.4.2 Mesh Split/Join

One strategy to balance the load of particles across mesh partitions is to simply split (geometrically) the overloaded partition into smaller pieces. After the particle-processing load on these new partitions has been reduced the caller may want to recombine the partitions to avoid unnecessary over-decomposition. The geometric rejoin poses a tough problem — *gjoin* has a long history of addressing just such a problem, and is still not the ideal solution.

Our solution is to avoid the geometric problem altogether by simply saving the mesh just before partitioning. To reconstitute a partitioned mesh one can simply read the saved, unsplit mesh partition from the data warehouse. The fields and particles must be handled by the caller since that is an application-specific operation.

One can envision similar techniques to archive application data before complex changes are made to the mesh or its associated data. In particular, when *rolling-back* to a previous task the data warehouse could archive that data needed to recreate the task in its original form. Though this likely is not viable for each task, it could provide a technique for a *check-pointing* of sorts. The term *hard-rollback* might be more appropriate. We seek to provide this capability and leave the decision of when use of this type of check-pointing to the PIC and AMT developers.

4.5 Data Warehouse Mechanics

All of the functionality above is based on some fundamental capabilities of the data warehouse. These are described below.

Key Management I An import role of the data warehouse is to insulate ATDM components from the existence of *keys* in Kelpie, the *key-value* store. While there are many data operations that manipulate *key-value* pairs, the user will be insulated from those operations and see a much simpler interface.

Key Management II Since Kelpie is a distributed key/value store, we can consider keys as lightweight proxies for much large data values. Using a $1D$ or $2D$ array, or a more complex graph, we can envision manipulating key relationships and then aggregating the data values into a composite data structure. A simple example is a $1D$ array partitioned into smaller pieces. A more complex example is a graph that describes the global partitioning scheme for the entire mesh.

App-aware Data Warehouse Application semantics are required for some data operations (e.g., storing fields to a file such that a different application can make sense of it later). This also allows for operations such as adding overlapping data as is done in the case of *ghost zones*.

App-agnostic Data Warehouse There may be cases where an application simply provides the data warehouse with a generic pointer to data — this is probably a good design choice in the early stages of applications development. The data warehouse will support these cases and provide a path to evolve them into a algorithm reusable to other data warehouse components.

Architecture Much of the data warehouse functionality can be handled by template meta-programming devices such as policies and traits. One advantage of this type of software design is mitigation of bloated APIs. They also allow a finer-grain control to the callers who can tailor a specific algorithm with data warehouse machinations handled by a host class. A more structured — and more rigid — architecture might emerge as the data warehouse matures, but the initial stages will benefit from the flexibility provided by modern template meta-programming techniques.

It easy to imagine more complex cases of topological relationships between data values are expressed as *keys* in a graph. For example, a top-level node may represent a specific field (e.g., "pressure"), and each second-level node represents a different *version* of that same field, with requisite sub-trees below the version nodes. *Version* here can mean timestep, refinement level, or whatever application-level semantics makes sense. Graph representations provide an elegant means of defining the data relationships.

It is important to note that any description of relations between *keys-value* pairs can itself be stored as a data object in data warehouse. This allows for a type of *self-describing* data. In practice, the caller can retrieve *meta-data* (i.e. a relation between keys) that can be used to locate the *key-value* pairs and find the required keys. The benefit is in reducing the burden of the application to coordinate task-data interactions and providing global data topologies in a lightweight data abject.

Since the data warehouse is so closely linked to the MeshDB in terms of data, and the AMT in terms of data operations, we hope to have significant co-design collaborations with both teams. In fact, we feel this is a requirement for a successful ATDM ecosystem.

It's clear that many algorithms for data management can be perform by algorithms contained in various ATDM components. We seek an abstraction layer that has the data warehouse responsible for operations in which most of the logic is concerned with data management, and other ATDM components responsible for algorithms that have little or no data management logic.

4.6 Third-party Consumers and Producers

Much work has been done in *in-situ* computing where the nodes participating in a physics computation also perform some type of post-processing. The *in-situ* approach is convenient for two reasons: (1) the datasets are large and we want to avoid significant communication for just one operation, and (2) the data produced by *in-situ* applications is closely related to the ATDM analysis. We argue that there are third-party applications that seek to process ATDM analysis data, but are not critical such that they be granted resources on a compute node. In those cases, Kelpie can be used as a conduit to provide data to third-party consumers without affecting the load on the compute nodes, modulo the I/O costs. Specific cases where third-party computing, or *cooperative computing*, is helpful are visualization and Reduced-Order Modeling (ROM).

The decision as to whether a post-processing application should be run using compute node resources or on machines that are not ATDM compute nodes should be made with a consideration of the problem at hand. As analysis needs change, so can the proximity of these additional operations.

Chapter 5

Data Warehouse Development

After gathering requirements from DHARMA and the application teams, the data warehouse team began evaluating our previous work with in-memory data stores for HPC to see how well it mapped to ATDM's needs. This evaluation primarily focused on three topics: APIs, performance, and usability. Following this examination, we began restructuring, adapting, and modernizing our prior work to make it more in line with what will be developed in FY16.

5.1 Identifying Areas for Improvement

The members of the data warehouse team have built a variety of communication software libraries in other ASC projects. These efforts were largely targeted at building scalable I/O services for HPC, and are therefore well-suited to data warehouse operations. However, given the new interfacing requirements listed in Chapters 3 and 4, we felt it was worthwhile to take a critical look at ways our software could be improved to make a more appropriate data warehouse for ATDM. The following topics were identified for Kelpie and Nessie, as well as our software development efforts in general:

Unexpected Overheads: One of the benefits of using Nessie is that it performs a good bit of the low-level communication work that is necessary for making distributed computing possible. For example Nessie automatically fragments large RPC arguments and employs a significant amount of intelligence to determine how data should be dynamically registered with the NIC for RDMA transfers. While these features make it easier to develop applications, they can lead to unexpected overheads that limit performance. These costs may vary significantly on different networks, leaving users confused as to why their software runs well on one architecture and poor on another. Dynamic registration was particularly problematic in our experiments. Given that ATDM components will have a more active role in data transfers, it is no longer necessary for the transport to handle *all* communication scenarios. It should instead be optimized for speed with predictable overheads.

Pinned Memory Reuse: Another anticipated problem with ATDM applications is that they will frequently need to allocate/deallocate small- to medium-sized blocks of memory. Other RDMA communication libraries such as GASNet[1] allocate a large region of contiguous memory at initialization time and then assign regions of the memory to different uses as the application progresses. This approach reduces the overhead for obtaining network accessible memory and makes smaller transfers more streamlined. A similar approach would be useful in the data warehouse.

Completion Notification: Kelpie and Nessie users currently perform asynchronous operations by issuing a request and querying the status of a request handle to determine when it has completed. This methodology works but forces the user to maintain a handle for the lifetime of an operation. The DHARMA team in particular is focused on callback styles of completion notification. As such, Kelpie will need to be restructured to perform completion callbacks in an efficient manner.

Streamlined RPC: While the NNTI layer has received a fair amount of attention over the last few years, there has been little development in the NSSI RPC layer outside of modifications made to improve C++ use. A holistic look at how the data warehouse and Kelpie use NSSI revealed a number of opportunities in which the transport software could be optimized.

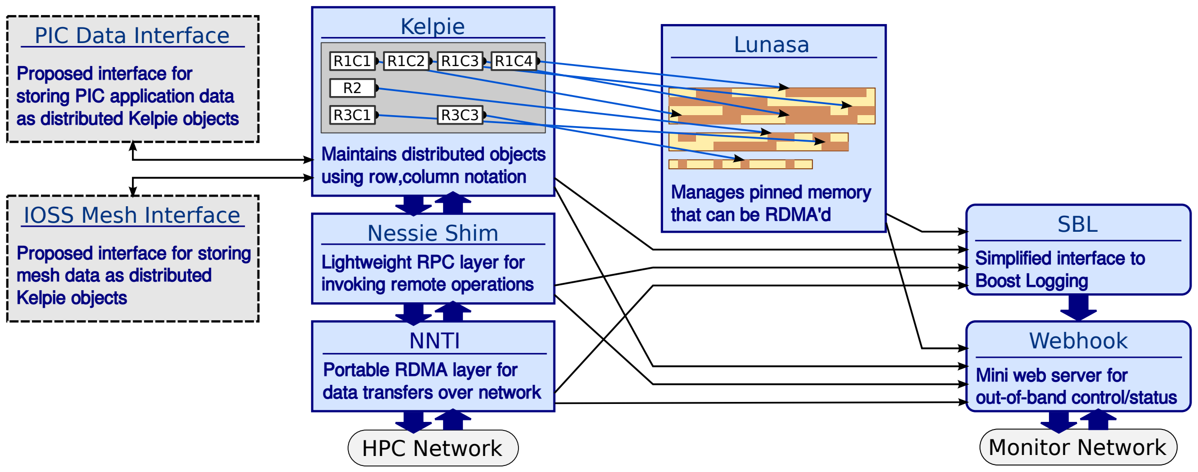
1. *Pinned Memory:* Kelpie handles the bookkeeping for pinned memory objects in the system and therefore never wants NSSI to locate or pin memory blocks on its own. Removing memory management functionality from NSSI could speed up the data path.
2. *Short Messages:* NSSI automatically handles arbitrary-length RPC inputs and outputs automatically through multiple data transfers. These operations impede performance and are easily overlooked by users. Kelpie would benefit from a more streamlined interface that forces hard limits on the maximum message size.
3. *C++ Aware:* NSSI is currently written in C, which limits Kelpie's ability to use some C++ features (e.g., class-based serialization and lambdas). It would be useful to have a C++-based transport.

Out-of-Band Control/Debug: The data warehouse will be constructed from a number of different software components, each with its own configuration and status settings. Based on our experiences with other complex data frameworks (e.g., Hadoop [6]), it is valuable to be able to remotely inspect the status of different components in a running application's software stack and manipulate various settings. A common library for providing a TCP-based interface would be extremely useful for the data warehouse.

Repository Accessibility: Nessie is currently distributed through the Trilinos software repository. While this packaging makes it easy for Sandia Trilinos users to acquire Nessie, it greatly increases the amount of effort new users must go through to get started with the software. In order to make data warehouse components more accessible, it is important that specific components be brought out from under Trilinos's umbrella and hosted independently in standalone software repositories that can be accessed more fluidly.

5.2 Improving Data Warehouse Components

The issues brought up during our self evaluation motivated us to begin rearchitecting our software to be better aligned for FY16 work. We took efforts to split our existing software into a number of independently-managed software repositories that can stand on their own. Each repository is hosted on Sandia's internal Gitlab site under the *nessie-dev* group[1].



PIC Data Interface
Proposed interface for storing PIC application data as distributed Kelpie objects
IOSS Mesh Interface
Proposed interface for storing mesh data as distributed Kelpie objects
Kelpie
R1C1
R1C2
R1C3
R1C4
R2
R3C1
R3C3
Maintains distributed objects using row,column notation
Nessie Shim
Lightweight RPC layer for invoking remote operations
NNTI
Portable RDMA layer for data transfers over network
HPC Network
Lunasa
Manages pinned memory that can be RDMA'd
SBL
Simplified interface to Boost Logging
Webhook
Mini web server for out-of-band control/status
Monitor Network

Figure 5.1. Data warehouse software was decomposed into multiple components.

The new organization of the data warehouse's different software components is illustrated in Figure 5.1. These components are summarized below.

NNTI 3.0: The Nessie Network Transport Interface (NNTI) provides a portable lightweight abstraction for RDMA operation on common HPC system interconnects. In addition to moving NNTI into its own repository, we refactored the API to be a more streamlined system that employs work requests to manage network operations.

NessieShim: NessieShim is a new, thin messaging layer that implements lightweight remote procedure calls (RPCs) on top of NNTI. This work took the essential portions of Nessie and reorganized them into a form that was better suited to C++.

Lunasa: Lunasa (pronounced Loo-NAH-sah) is a new project that implements a lightweight memory management unit for NNTI. This unit requests large regions of pinned memory from NNTI and then divides the allocations out to end applications as needed. By allocating in advance and reusing memory, the system helps the data warehouse avoid overheads that were observed in Nessie when memory was dynamically allocated.

[1] `https://gitlab.sandia.gov/groups/nessie-dev`

Kelpie: Kelpie is an in-memory data store that allows users to move data objects between nodes in a flexible manner. Kelpie's developments for this year are discussed in detail in Chapter 2.

Webhook: Webhook is a flexible out-of-band communication service that allows software components in the data warehouse to be queried and configured over a standard HTTP connection. Webhook implements a trivial web server and enables application components to register callback operations that can respond to specific queries by end users.

SBL: SBL is a Simplified Interface to Boost.Log. We conducted a survey of common third party logging libraries and determined that Boost.Log provided the best fit for our work. SBL was implemented as a simple wrapper around Boost.Log to make it easier for different data warehouse components to incorporate its logging functionality in a standard way.

The *PIC Data Interface* and *IOSS Mesh Interface* are the first data warehouse interfaces that have been defined for ATDM. Planning for these components was discussed in Chapter 4. The components will be developed in FY16.

More detailed activities for each of these efforts are provided as follows.

5.2.1 NNTI 3: A Portable RDMA Transport for HPC

The Nessie Network Transport Interface (NNTI) provides a portable, lightweight abstraction for RDMA operations on common HPC systems. Our current implementation includes support for the InfiniBand, Cray Gemini, and IBM BG/Q interconnects. The API includes commands to open/close the interface, connect/disconnect a peer, register/deregister memory buffers, send/receive messages, and transport bulk data asynchronously (put, get, and wait). The NNTI library was originally developed as part of Sandia's Nessie RPC project to enable portability across HPC interconnects. NNTI is built around four core concepts: memory buffers, send operations, RDMA operations, and events.

Memory Buffers: Many HPC interconnects require memory regions to be registered with the NIC before the memory can be used in data transfers. In order to do DMA, the NIC must know the physical address of the memory region involved. When the application registers the memory region, the pages are pinned to prevent the VMM from relocating the pages and changing the physical addresses. NNTI tracks these memory regions and provides the application with a handle that can be shared with peers to perform data operations.

Send Operations: The NNTI send protocol is a messaging protocol used to transfer data from sender to receiver. The protocol uses command packets to initiate the transfer and tell the receiver the parameters of the message including destination, length and event flags. The exact format of the command packet is transport specific, but it is expected that it contains enough information for the receiver to make decisions about message delivery.

RDMA Operations: The NNTI RDMA API is a lightweight one-sided API that is mapped as closely as possible to the interconnect with native one-sided operations. On interconnects that do not have native one-sided operations, NNTI uses a protocol similar to the send protocol to manage the transfer.

Events: All NNTI data transfer operations are asynchronous. NNTI events are generated at the completion of data operations and contain the detailed results of the operation. Completion does not mean success, so the event's result field must be check for each operation. Event delivery is selected by the initiator when the operation is submitted. Events can be delivered at the initiator, the target, neither or both.

NNTI is currently undergoing an API revision [4] to better support dynamic programming models. As part of this work, the internals are being rewritten in C++ which gives NNTI a native C++ interface with a C wrapper. This is the reverse of previous versions which only had a C API that made some programming tasks (e.g., callbacks) tedious in C++.

5.2.2 NSSI Shim: A Thin RPC Layer for NNTI

Nessie's existing NSSI layer provides a good, general-purpose remote procedure call (RPC) library that makes it easy for users to invoke data processing operations on remote nodes. While this layer is flexible enough to be used in a variety of ways, we have observed that NSSI's generality can impede performance in certain workloads. In situations where higher-level services such as Kelpie only use a fraction of the RPC library's capabilities, it is useful to consider optimizations to the RPC library's data path that would improve performance.

NssiShim is a lightweight version of NSSI that is focused on implementing Kelpie's RPC operations as efficiently as possible. It is a complete redesign of Nessie. Key differences are characterized as follows:

C++: NSSI was largely written in C, which at times made it difficult to use from C++. NssiShim was written in C++ to provide better coupling with other components in the data warehouse.

User-Managed Data Buffers: NSSI transparently handles memory registration for its users and employs APIs that only use virtual addresses to reference memory. Given that Kelpie is responsible for managing chunks of registered data, NssiShim expects most messages to contain only RDMA pointers that RPCs will use to invoke additional data transfers. This design point places greater responsibility on the user, but also greatly simplifies the RPC layer.

Single-Message RPCs: NSSI allows users to transfer a variable amount of data for both inputs to and outputs from an RPC. While this feature makes RPCs easier to use, they can result in additional processing delay in the library, as well as invoke multiple data transfers. NssiShim only supports RPCs that pack their arguments in single messages. This minimalist style fits the expectation that services such as Kelpie will largely be sending messages with RDMA pointers, and that they will manage the buffers themselves.

Simplified Serialization: The original NSSI implementation used XDR for data serialization. While XDR is a quick and rugged serialization library, its C interfaces can be difficult to work with from C++ software. NssiShim provides a simple serialization interface that allows users to append their data structures into the end of a message buffer. Custom serialization operations can easily be added to classes. Lambda operators make it possible to perform serialization without intermediate buffers.

A deeper understanding of how NssiShim is architected can be better illustrated by tracing through the different operations that take place during an RPC:

Send a Request: All RPC calls are initiated by a client that wants to invoke a specific function on a remote node. The NssiShim `call` function selects the next available outgoing message buffer and begins assembling header information into it based on the user's specifications. The `call` function appends the RPC's arguments onto the back of the buffer based on a packing function supplied by the user. The `call` operation aborts with an error if the packing would overflow the buffer, as the expectation is that a user should never expect a `call` operation to fragment into multiple messages. A result buffer is also acquired at this time to provide a place for the remote node to return a short reply for the transaction. The message is transmitted to its destination via NNTI's `send` message operation.

Request Buffer Deallocation: Once NNTI finishes transmitting the request message, it triggers an event notifying NssiShim that the send of the request buffer has completed locally and the request is now pending processing on the remote side. NssiShim can then mark the request buffer as available for subsequent messages.

Message Receival: When the message arrives at the remote NNTI queue, it triggers an event that is handled by NssiShim. NssiShim unpacks the RPC request header and invokes the corresponding RPC function to process the message. NssiShim *does not* move any bulk data that may be associated with the message automatically. Instead, a small amount of RPC data (such as an RDMA buffer handle) can be embedded in the message and then retrieved by the RPC using NNTI operations.

Reply with Optional Results: An RPC may send a reply back to the sender that includes optional result data of limited size. NssiShim allocates a buffer for assembling the reply message and then transmits the message as a `put` operation. Once the `put` completes, the reply message's buffer space is deallocated.

Sender Completion: The arrival of a reply message with optional results triggers an event in NNTI that is processed by NssiShim. Replies are currently detected and observed by user applications through `wait` operations.

The current implementation of NssiShim follows the same `call/wait` nonblocking semantics as the original NSSI layer. Given that DHARMA prefers asynchronous messages be specified entirely at issue time, NssiShim will be adapted in the near future to allow users to define a callback to be invoked when the RPC completes. This callback should streamline message processing, as users will no longer need to poll for a message's completion.

5.2.3 Lunasa: Memory Allocator for RDMA Data

Lunasa (pronounced Loo-NAH-sah) is a memory allocator written on top of the NNTI transport library. Kelpie and the data warehouse use of NNTI may result in the allocation of many small blocks of pinned memory for use in transport. These many small allocations result in poor performance. Lunasa is an approach similar to malloc and virtual memory paging. It allocates a large block of NNTI memory upfront and as needed. As requests for smaller memory come in, it allocates and tracks offsets into this larger block of memory. The transport layer operates as well on offsets and length as the whole block, so this allows us to avoid making too many of the high cost allocations at any time. As offsets are freed, Lunasa returns the blocks to its pool of available memory for later reallocation.

Lunasa uses a linked-list, greedy allocator. It keeps track of both filled and empty regions sorted in order of their offsets into the page. When a new buffer is allocated, the first available, large enough offset will be taken and the memory assigned. The remainder, if any, becomes a new free region.

Currently there are no options for aligning the allocated offsets. If the available free regions are not large enough to support the request, then a new page will be allocated. When memory is freed any adjoining regions will be merged into a single freed region. However, fragmentation is still inevitable. In experiments this tends to be no more than 10 percent. However, it remains to be seen how much overhead occurs in practice.

To aid understanding these overheads, Lunasa exports its current state as a Webhook function allowing users to monitor the current allocations and available regions. This allows us to monitor, through a webpage, how well Lunasa performs as it is running.

Because this is a generalized allocator for NNTI memory, it turns out to be relatively easy to make it a Kokkos allocation space as well. Kokkos' space API expects only allocation and deallocation through void pointers. By implementing a wrapper around the Lunasa calls in the expected Kokkos API, we are able to support a vector created directly in pinned memory, like:

```
View<double[10], LunasaSpace> myVector;
```

This capability allows us to manage Kokkos vectors across the entire system through Kelpie and the data warehouse, where Kokkos manages memory exclusively on the node.

While the linked-list performance is acceptable, some optimizations in future work may be considered, especially caches for common allocation sizes, usually powers of 2. This optimization keeps a cache in a set of buckets, labeled by the common sizes, of currently available freed regions that can fit such sizes.

5.2.4 Webhook: A Reusable Debug/Control Interface

Distributed software applications are often challenging to debug, monitor, and tune because it is difficult to access the inner details of a particular software component without substantial instrumentation. For example, when Kelpie was first being developed, we spent a significant amount of time debugging how nodes exchanged metadata about each other. While logging provided a history of what different nodes in the system were doing, it took a good bit of manual analysis to assemble all of the logs together and verify the nodes had the right states at the right times. What we wanted was a way that we could more easily examine the state of running nodes and set different variables if needed.

System software developers in the big data community have faced similar problems, and routinely utilize an appealing solution: they embed a lightweight web server in a software component that allows users to query and manipulate the component as it runs. This approach is beneficial because users can interact with their software using either a web browser or command line tools. Application developers typically utilize an existing web server to handle HTTP requests, and then implement functions that respond to specific queries (usually in a RESTful API manner).

In order to make such services available for the data warehouse, we constructed a standalone library called *Webhook*. This library launches a simple webserver thread for an application that can be used by different software components. The webserver uses Boost's asynchronous I/O (asio) module and is largely based on the http reference design by Christopher M. Kohlhoff. A Webhook user defines one or more hooks they would like the node to perform. A hook is composed of two parts: a name for the operation "/mycomponent/status" and a function to invoke when a request for that operation is received. Webhook handles most of the parsing work for requests. A user's function is handed a map of all the input arguments that were provided and expects a string that provides all the data needed in the response.

Our early experiences with Webhook have found that it has uses beyond debugging. One problem that we have had for years with NNTI is resolving new connection requests. The original code used a non-blocking TCP socket to handle new requests for connections. Unfortunately, the implementation would occasionally have timing issues that caused pairs of nodes to deadlock on each other without any warning to the user. Changing this operation to be performed by a Webhook has multiple benefits: the sockets software is no longer needed in NNTI, the operation can be performed asynchronously without blocking other functions, and one socket can be easily used by different components in the application's stack.

5.2.5 SBL: A Simplified Interface to Boost.Log

In previous implementations of Nessie, a custom logger was constructed that allowed the software to capture internal messages at different layers in the stack. While this software worked well, it was oriented at C code and was not used outside of applications that directly used Nessie. We explored different open source logging packages that are currently available. Boost's Boost.Log library was selected for it's stability, wide availability and flexibility. Flexibility comes at the cost of a large API with a learning curve to match. SBL was developed to balance the flexibility and complexity. It is composed of two simple classes that consolidate Boost.Log's core features:

`sbl::source` Source is a log message producer that attaches a severity and channel attribute to each message. SBL defines five severity levels from debug (lowest) to fatal (highest). Each sbl::source object is assigned a severity at construction that can be changed during the life of the object. The channel attribute is an application defined string that groups together related log messages. The channel attribute is not assigned to the `sbl::source`, but instead is assigned to each message when it is produced. If a channel is not assigned by the application, the message will be placed in the global channel. As each message is produced, it is broadcast to all `sbl::stream` objects.

`sbl::stream` Stream is a log message consumer that receives messages from all `sbl::source` objects. Each `sbl::stream` has an `std::ostream` where log messages get written. `sbl::stream` can open a file by name, write to `std::clog` or write to any `std::ostream` that the application has previously created. As each message is received, a severity filter is applied that determines if the message should be written to the underlying stream. The severity filter uses a channel map to give each channel its own severity with a fallback severity for messages in the global channel. The severity filter rejects messages with a severity lower than the severity of the channel.

Multiple sources and streams can interact in unexpected ways that can lead to developer frustration. So for the simplest cases where a single stream of log messages is the basic requirement, `sbl::logger` was created. `sbl::logger` is a further simplification of Boost.Log that combines multiple `sbl::source`'s and an `sbl::stream` into a single interface. `sbl::logger` uses additional attributes and a more restrictive filter to hide its messages from other `sbl::stream` objects and block out messages from other `sbl::source` objects. While not as flexible as separate sources and streams, this usage may feel more natural to users familiar with simpler loggers.

SBL is fully implemented and ready for use. There are a number of possible options for future work. SBL does not have any special severity levels that would force an `sbl::stream` to accept or reject a message. These special levels are available is some other logging libraries and could be added to SBL if there is interest from developers. SBL does not support removal of a channel from the channel filter. To gain the same effect, the application can set the severity for the channel to match the severity of the global filter.

References

[1] Dan Bonachea. Gasnet specification, v1. 1. 2002.

[2] H Carter Edwards, Christian R Trott, and Daniel Sunderland. Kokkos: Enabling many-core performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202–3216, 2014.

[3] Francis H Harlow and MW Evans. A machine calculation method for hydrodynamic problems. *LAMS-1956*, 1955.

[4] Todd Kordenbrock. The NNTI 3.0 programming interfac.

[5] 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.

[6] Tom White. *Hadoop: The definitive guide.* " O'Reilly Media, Inc.", 2012.

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)

Sandia National Laboratories