

|  |  |
| --- | --- |
| **Date:March 21, 2013** | **High Level Design – Function Shipper****FOR EXTREME-SCALE COMPUTING RESEARCH AND DEVELOPMENT (FAST FORWARD) STORAGE AND I/O** |

|  |  |
| --- | --- |
| LLNS Subcontract No. | B599860 |
| Subcontractor Name | Intel Federal LLC |
| Subcontractor Address | 2200 Mission College Blvd.Santa Clara, CA 95052 |

|  |
| --- |
| **Limited Rights Notice.  These data are submitted with limited rights under Prime Contract No. DE-AC52-07NA27344 between LLNL and the Government and Subcontract No. B599860 between LLNL and Intel Federal LLC.  This data may be reproduced and used by the Government with the express limitation that it will not, without written permission of Intel, be used for purposes of manufacture nor disclosed outside the Government.****The information contained herein is confidential and proprietary, and is considered a “trade secret” under 18 U.S.C. § 1905 (The Trade Secrets Act) and Exemption 4 to FOIA.  Release of this information is prohibited.** |

Table of Contents

[Introduction 1](#_Toc351675905)

[Definitions 1](#_Toc351675906)

[Changes from Solution Architecture 1](#_Toc351675907)

[Specification 2](#_Toc351675908)

[Overview 2](#_Toc351675909)

[Metadata and Generic Function Shipping 3](#_Toc351675910)

[Bulk Data Transfers 4](#_Toc351675911)

[MPI Network Plugin 5](#_Toc351675912)

[API and Protocol Additions and Changes 5](#_Toc351675913)

[Network Abstraction Layer API 6](#_Toc351675914)

[Generic Processor Macros 7](#_Toc351675915)

[Function Shipper API (client) 8](#_Toc351675916)

[Function Shipper handler API (server) 8](#_Toc351675917)

[Bulk Data Shipper API 9](#_Toc351675918)

[Open Issues 10](#_Toc351675919)

[Large metadata transfers 10](#_Toc351675920)

[Risks & Unknowns 10](#_Toc351675921)

**Revision History**

|  |  |  |
| --- | --- | --- |
| **Date** | **Revision** | **Author** |
| March 21, 2013 | 1.0 | Jerome Soumagne, The HDF Group |
| March 21, 2013 | 1.1 | Jerome Soumagne, Quincey Koziol, The HDF Group |
|  |  |  |
|  |  |  |

# Introduction

High performance I/O on exascale systems is not expected to be feasible without exporting the I/O API from I/O nodes onto the compute nodes. One solution to address this problem is to use a method called function shipping. Making use of this method, I/O calls issued from the compute nodes are locally encoded, sent through the network to the I/O nodes where they in turn get decoded and executed—with the operation’s result being sent back to the issuing node. This document describes the implementation of the function shipping framework implemented as part of our FastForward project.

# Definitions

CN – compute node

ION – I/O node

RMA – remote memory access

# Changes from Solution Architecture

There is no change from the initial design, the framework has been implemented and refined to follow what was originally presented.

# Specification

As described in the 2.4 design document (Function Shipping Design and Framework Demonstration), the function shipper framework is derived from the I/O Forwarding Scalability Layer (IOFSL) to follow a more generic and transport independent approach: the interface is designed to be as generic as possible to allow any I/O function call to be forwarded to the I/O nodes; and the network implementation is abstracted so that alternate mechanisms can be implemented and selected, making use of the various transport mechanisms available on the system.

## Overview

The function shipper follows a client/server architecture. The client ships calls asynchronously and returns back to the application while it waits for their completion, the server receives these calls, executes them and sends the response back to the client. Input and output parameters of the function calls are serialized (or encoded) so that they can be easily transferred across the network.

**Function Shipper Client**

**Function Shipper Server**

*Compute Node*

*I/O Node*

Metadata

Bulk data

Figure 1. Client/server architecture.

The function shipper client is located on a compute node, the function shipper server on an I/O node. A given function shipper client may communicate with different function shipper servers. Function shipper servers may be launched independently and do not communicate with each other in the current design.

As one can see in Figure 1, we consider two types of transfers for shipping I/O function calls: metadata and bulk data transfers. To give flexibility to the user and allow transfers to be as efficient as possible, the function shipper framework is divided into two separate interfaces, one that initiates remote function calls and forwards/receives metadata information (two-sided communication) and one that initiates bulk data transfers and handles remote memory accesses (one-sided communication).

Figure 2 represents the function shipper software stack (for both the client and the server). One can see in Figure 2 that both the function shipper and the bulk data shipper interfaces are built on top of the same network abstraction layer. The network abstraction layer hides the network interface from the application and allows multiple network protocols to be dynamically selected. It can provide both point-to-point and remote memory access operations.

Network Interface Controller

**Bulk Data Shipper**

**Function Shipper**

**Network Abstraction Layer**

MPI

BMI

…

Figure 2. Function shipper interfaces.

As a consequence, the function shipper interface and the bulk data shipper interface may select a specific network protocol so that metadata and bulk data can be transferred in an efficient manner (which may not be necessarily the same: e.g., if the function shipper interface makes use of the BMI plugin, one can make use of the MPI plugin to perform bulk data transfers).

In the following sections we only consider the MPI plugin as the network abstraction layer plugin. Additional plugins, such as a Lustre Networking (LNET) plugin, may be added in the future.

## Metadata and Generic Function Shipping

Shipping a function call to the function shipper server means that the client must know how to encode and decode the input and output parameters before it can start sending information. On the server side, the function shipper server must also have knowledge of what function to execute when it receives a call and how it can decode and encode the input and output parameters. This framework for describing the function calls and encoding/decoding parameters is key to the operation of the function shipper.

One of the requirements of the function shipping framework is the ability to support the set of function calls that can be shipped to the server in a generic fashion, avoiding the limitations of a hard-coded set of routines to ship. The generic encode/decode framework is described in Figure 3. During the initialization phase, the client and server register encoding and decoding functions by using a unique function name that is mapped to a unique ID for each operation, shared by the client and server. The server also registers the callback that needs to be executed when an operation ID is received with a function call. To send a function call that does not involve bulk data transfer, the function shipper client encodes the input parameters along with that operation’s ID into a buffer and send it to the client asynchronously using an unexpected messaging protocol.

This therefore limits the message size to the size of an eager message (i.e., a few kilobytes). Note that to ensure full asynchrony, the memory buffer used to receive the response back from the server is also pre-posted by the client.

**Function Shipper Server**

**Function Shipper Client**

***1. Register encoding / decoding functions into mapping table and get ID***

***3. Decode / Get input parameters / Execute function / Set output parameters / Encode / Send response***

***1. Register encoding / decoding functions and function callback into mapping table***

***2. Pass ID + input parameters/ Encode / Send operation***

Figure 3. Metadata and generic function shipping.

When the server receives a new operation ID, it looks up the corresponding callback, decodes the input parameters, executes the function call, encodes the output parameters and sends the response back to the client. Note that this response may also be sent asynchronously. While receiving new function calls, the server also tests the list of response requests to check their completion, freeing the corresponding resources when an operation completes.

Once the client has knowledge that the response has been received (using a wait/test call) and therefore that the function call has been remotely completed, it can decode the output parameters and free the resources that were used for the transfer.

## Bulk Data Transfers

In addition to the previous mechanism, some function calls may require the transfer of larger amounts of data. For these function calls, the bulk data shipper interface is used.



Figure 4. Bulk data transfers (“write” operation execution case).

As described in Figure 4, the bulk data transfer interface uses a one-sided communication approach. The function shipper client exposes a memory region to the function shipper server by creating a bulk data descriptor (which contains memory address information, size of the memory region that is being exposed, and other parameters that depend on the underlying network implementation). This bulk data descriptor can then be serialized and shipped to the function shipper server along with the function call input parameters. When the server decodes the input parameters it deserializes the bulk data descriptor and gets the size of the memory buffer that has to be transferred.

In the case of a *write* operation, the function shipper server may allocate a buffer of the size of the data that needs to be received, expose the memory region by creating a bulk data block descriptor and initiate an asynchronous read/get operation on that memory region. The function shipper server then waits for the completion of the operation and executes the *write* call once the data has been fully received. The response (i.e., the result of the *write* call) is then sent back to the function shipper client and memory handles are freed.

In the case of a *read* operation, the function shipper server may allocate a buffer of the size of the data that needs to be read, expose the memory region by creating a bulk data block descriptor, execute the *read* call, then initiate an asynchronous write/put operation to the client memory region that has been exposed. The function shipper server may then wait for the completion of the operation and send the response (i.e., the result of the *read* call) back to the function shipper client. Memory handles can then be freed.

## MPI Network Plugin

Network operations previously described are built on top of a network abstraction layer. To demonstrate the functionality of the function shipping framework, an MPI plugin has been developed and implements the network abstraction layer described below.

The MPI plugin implements two-sided transfers (unexpected and expected messaging) using non-blocking two-sided operations.

For one-sided transfers (i.e., bulk data transfers), it is important to note that this plugin implements one-sided communication on top of two-sided due to the lack of flexibility of the MPI 2 RMA specification (although it is also being tested using MPI 3 functionality). Progress must therefore be made on the client whenever a bulk data operation needs to be realized and this is done currently by using a thread (other plugins implementing the bulk data interface should not and are not expected to use a thread on the client side to avoid overheads in user applications). Plugins built on top of the Lustre Networking layer or Cray Gemini layer (supporting RMA operations natively) will not have this limitation.

To be able to launch client and server separately and simulate a normal usage of the interface, the MPI dynamic connection interface is used. This is also not a suitable solution for large systems as dynamic process management is not supported, and will be replaced by another mechanism when available.

# API and Protocol Additions and Changes

The 2.4 design document (Function Shipping Design and Framework Demonstration) introduced the network abstraction layer. We describe here one modification realized to that API, as well as the higher level function shipper and bulk data shipper APIs.

## Network Abstraction Layer API

The only change in this API is the *network\_class* pointer added to every function call, giving the user the ability to use different network plugins at the same time (which requires their initialization) and therefore optimizes the communication depending on the size of the data that is to be transferred.

/\* Initialize the MPI plugin \*/

na\_network\_class\_t \*na\_mpi\_init(MPI\_Comm \*intra\_comm, int flags);

/\* Finalize the network abstraction layer \*/

int na\_finalize(na\_network\_class\_t \*network\_class);

/\* Get the maximum size of an unexpected message \*/

na\_size\_t na\_get\_unexpected\_size(na\_network\_class\_t \*network\_class);

/\* Lookup an addr from a peer address/name \*/

int na\_addr\_lookup(na\_network\_class\_t \*network\_class,

 const char \*name, na\_addr\_t \*addr);

/\* Free the addr from the list of peers \*/

int na\_addr\_free(na\_network\_class\_t \*network\_class,

 na\_addr\_t addr);

/\* Send a message to dest (unexpected asynchronous) \*/

int na\_send\_unexpected(na\_network\_class\_t \*network\_class,

 const void \*buf, na\_size\_t buf\_len, na\_addr\_t dest,

 na\_tag\_t tag, na\_request\_t \*request, void \*op\_arg);

/\* Receive a message from source (unexpected asynchronous) \*/

int na\_recv\_unexpected(na\_network\_class\_t \*network\_class,

 void \*buf, na\_size\_t \*buf\_len, na\_addr\_t \*source,

 na\_tag\_t \*tag, na\_request\_t \*request, void \*op\_arg);

/\* Send a message to dest (asynchronous) \*/

int na\_send(na\_network\_class\_t \*network\_class,

 const void \*buf, na\_size\_t buf\_len, na\_addr\_t dest,

 na\_tag\_t tag, na\_request\_t \*request, void \*op\_arg);

/\* Receive a message from source (asynchronous) \*/

int na\_recv(na\_network\_class\_t \*network\_class,

 void \*buf, na\_size\_t buf\_len, na\_addr\_t source,

 na\_tag\_t tag, na\_request\_t \*request, void \*op\_arg);

/\* Register memory for RMA operations \*/

int na\_mem\_register(na\_network\_class\_t \*network\_class,

 void \*buf, na\_size\_t buf\_len, unsigned long flags,

 na\_mem\_handle\_t \*mem\_handle);

/\* Deregister memory \*/

int na\_mem\_deregister(na\_network\_class\_t \*network\_class,

 na\_mem\_handle\_t mem\_handle);

/\* Serialize memory handle for exchange over the network \*/

int na\_mem\_handle\_serialize(na\_network\_class\_t \*network\_class,

 void \*buf, na\_size\_t buf\_len, na\_mem\_handle\_t mem\_handle);

/\* Deserialize memory handle \*/

int na\_mem\_handle\_deserialize(na\_network\_class\_t \*network\_class,

 na\_mem\_handle\_t \*mem\_handle, const void \*buf, na\_size\_t buf\_len);

/\* Free memory handle \*/

int na\_mem\_handle\_free(na\_network\_class\_t \*network\_class,

 na\_mem\_handle\_t mem\_handle);

/\* Put data to remote target \*/

int na\_put(na\_network\_class\_t \*network\_class,

 na\_mem\_handle\_t local\_mem\_handle, na\_offset\_t local\_offset,

 na\_mem\_handle\_t remote\_mem\_handle, na\_offset\_t remote\_offset,

 na\_size\_t length, na\_addr\_t remote\_addr, na\_request\_t \*request);

/\* Get data from remote target \*/

int na\_get(na\_network\_class\_t \*network\_class,

 na\_mem\_handle\_t local\_mem\_handle, na\_offset\_t local\_offset,

 na\_mem\_handle\_t remote\_mem\_handle, na\_offset\_t remote\_offset,

 na\_size\_t length, na\_addr\_t remote\_addr, na\_request\_t \*request);

/\* Wait for a request to complete or until timeout (ms) is reached \*/

int na\_wait(na\_network\_class\_t \*network\_class,

 na\_request\_t request, unsigned int timeout, na\_status\_t \*status);

## Generic Processor Macros

To automatically generate encoding and decoding functions, we make use of the BOOST preprocessor subset that is able to operate on a sequence of given elements. Encoding or decoding operations are very similar operations, we can therefore use the same *processor* function to encode or decode parameters.

The macro prototype is given below:

/\* IOFSL\_SHIPPER\_GEN\_PROC( struct\_type\_name, fields ) \*/

Generating the *processor* function and corresponding structure to send an integer would require the following macro:

IOFSL\_SHIPPER\_GEN\_PROC( function\_in\_t, ((int32\_t)(func\_param1)) )

This would generate the following code:

/\* Define function\_in\_t \*/

typedef struct {

 int32\_t func\_param1;

} function\_in\_t;

/\* Define fs\_proc\_function\_in\_t \*/

static inline int fs\_proc\_function\_in\_t(fs\_proc\_t proc, void \*data)

{

 int ret = S\_SUCCESS;

 function\_in\_t \*struct\_data = (function\_in\_t \*) data;

 ret = fs\_proc\_int32\_t(proc, &struct\_data->func\_param1);

 if (ret != S\_SUCCESS) {

 S\_ERROR\_DEFAULT("Proc error");

 ret = S\_FAIL;

 return ret;

 }

 return ret;

}

Note that the size of the integer needs to be explicitly stated to avoid encoding/decoding errors if integer sizes differ between the function shipper client and the function shipper server.

Additional types to support filenames (fs\_string\_t) and bulk data handles (bds\_handle\_t) can be passed to these macros. More complex structures require definition of the substructures and a call to this macro to generate the specific *processor* functions.

## Function Shipper API (client)

The function shipper API is quite straightforward. Note that the fs\_forward function call allows network abstraction (na\_addr\_t) *addresses* to be passed, which describe the network address of the remote function shipper server. Therefore multiple I/O nodes can be selected and their address passed to the function shipper layer. This address can be retrieved using the network abstraction layer.

The following routines compose the client API:

/\* Initialize the function shipper and select a network protocol \*/

int fs\_init(na\_network\_class\_t \*network\_class);

/\* Finalize the function shipper \*/

int fs\_finalize(void);

/\* Register a function name and provide a unique function identifier \*/

fs\_id\_t fs\_register(const char \*func\_name,

 int (\*enc\_routine)(fs\_proc\_t proc, void \*in\_struct),

 int (\*dec\_routine)(fs\_proc\_t proc, void \*out\_struct));

/\* Forward a call to a remote server \*/

int fs\_forward(na\_addr\_t addr, fs\_id\_t id,

 const void \*in\_struct, void \*out\_struct, fs\_request\_t \*request);

/\* Wait for an operation request to complete \*/

int fs\_wait(fs\_request\_t request, unsigned int timeout, fs\_status\_t \*status);

/\* Wait for all operations to complete \*/

int fs\_wait\_all(int count, fs\_request\_t array\_of\_requests[],

 unsigned int timeout, fs\_status\_t array\_of\_statuses[]);

## Function Shipper handler API (server)

The function shipper handler is only used on the server. The main fs\_handler\_process routine receives new function calls, decodes the function operation ID and executes the callback that corresponds to that ID. This callback, which is manually defined for now (but can be automatically generated as well), will typically call fs\_handler\_get\_input and fs\_handler\_complete in addition to performing the remote operation. Note that the completion call will be asynchronous in the future to allow asynchronous response.

The following routines compose the server API:

/\* Initialize the function shipper handler and select a network protocol \*/

int fs\_handler\_init(na\_network\_class\_t \*network\_class);

/\* Finalize the function shipper handler \*/

int fs\_handler\_finalize(void);

/\* Register a function name that can be remotely executed \*/

void fs\_handler\_register(const char \*func\_name,

 int (\*fs\_routine) (fs\_handle\_t handle),

 int (\*dec\_routine)(fs\_proc\_t proc, void \*in\_struct),

 int (\*enc\_routine)(fs\_proc\_t proc, void \*out\_struct));

/\* Get input from handle \*/

int fs\_handler\_get\_input (fs\_handle\_t handle, void \*in\_struct);

/\* Get remote addr from handle \*/

const na\_addr\_t fs\_handler\_get\_addr (fs\_handle\_t handle);

/\* Receive a call from a remote client and process it \*/

int fs\_handler\_process(unsigned int timeout);

/\* Forward the response back to the remote client and free handle \*/

int fs\_handler\_complete(fs\_handle\_t handle, const void \*out\_struct);

## Bulk Data Shipper API

The bulk data shipper API is used on both the server and the client, although only the server will initiate transfers. The client only uses the first functions (bds\_handle\_create, bds\_handle\_free, bds\_handle\_serialize) to create a bulk data handle and send it to the function shipper server. The function shipper server uses the other functions to get/put the data to the local/remote memory location. Note that only contiguous memory regions are supported by the current version of the API. Non-contiguous regions will be supported in the next milestone (and can already be supported by the network abstraction layer).

The following routines compose the bulk data shipper API:

/\* Initialize the bulk data shipper and select a network protocol \*/

int bds\_init(na\_network\_class\_t \*network\_class);

/\* Finalize \*/

int bds\_finalize(void);

/\* Create bulk data handle from buffer (register memory, etc) \*/

int bds\_handle\_create(void \*buf, size\_t buf\_len, unsigned long flags,

 bds\_handle\_t \*handle);

/\* Free bulk data handle \*/

int bds\_handle\_free(bds\_handle\_t handle);

/\* Get data size from handle \*/

size\_t bds\_handle\_get\_size(bds\_handle\_t handle);

/\* Serialize bulk data handle into buf \*/

int bds\_handle\_serialize(void \*buf, na\_size\_t buf\_len, bds\_handle\_t handle);

/\* Deserialize bulk data handle from buf \*/

int bds\_handle\_deserialize(bds\_handle\_t \*handle, const void \*buf, na\_size\_t buf\_len);

/\* Write data \*/

int bds\_write(bds\_handle\_t handle, na\_addr\_t dest, bds\_block\_handle\_t block\_handle);

/\* Read data \*/

int bds\_read(bds\_handle\_t handle, na\_addr\_t source, bds\_block\_handle\_t block\_handle);

/\* Wait for bulk data operation to complete \*/

int bds\_wait(bds\_block\_handle\_t block\_handle, unsigned int timeout);

/\* Create bulk data handle from buffer (register memory, etc) \*/

int bds\_block\_handle\_create(void \*buf, size\_t block\_size, unsigned long flags,

 bds\_block\_handle\_t \*handle);

/\* Free block handle \*/

int bds\_block\_handle\_free(bds\_block\_handle\_t block\_handle);

/\* Get data size from block handle \*/

size\_t bds\_block\_handle\_get\_size(bds\_block\_handle\_t block\_handle);

# Open Issues

## Large metadata transfers

It is worth noting that the function shipper makes use of unexpected messaging to send metadata over the network. This therefore limits the size of the buffers that can be shipped. If the metadata that needs to be shipped exceeds the size of an unexpected message, the function shipper client will need to transfer the metadata in separate messages, making use of the bulk data shipper (using the previously described mechanism) to send the additional metadata to the function shipper server.

The same problem may appear for the response, as the receive buffer is pre-posted. This may force the function shipper server to wait for the client to post a new receive before the response can be transmitted (although sending response asynchronously should solve that issue in the future).

# Risks & Unknowns

The main unknown currently is the final network abstraction layer implementation, which should be addressed in the next milestone so that a specific network plugin can be developed, which will implement both unexpected messaging and RMA transfers efficiently. We believe this will be implemented with the Lustre networking (LNET) protocol, but a user-space implementation of LNET is not yet available.