Date: 2013-09-26	**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

Table of Contents

Introduction 1

Definitions 1

Changes from Solution Architecture 1

Specification 2

- Overview 2
- Metadata and Generic Function Shipping 3
- Bulk Data Transfers 4
- Network Plugins 5

API and Protocol Additions and Changes 6

- Network Abstraction Layer API 6
- Generic Processor Macros 8
- Function Shipper API (client) 8
- Function Shipper handler API (server) 9
- Bulk Data Shipper API 10
- Abstract checksums and Checksum API 12

Open Issues 13

- Bulk data transfers 13

Risks & Unknowns 13

Revision History

Date	Revision	Notes	Author
March 21, 2013	1.0		Jerome Soumagne, The HDF Group
March 21, 2013	1.1	Delivered to DOE as part of Milestone 3.3	Jerome Soumagne, Quincey Koziol, The HDF Group
June 20, 2013	2.0	Modifications include BMI plugin, non-contiguous bulk data transfers, adoption of Mercury as name (also reflected in APIs), and Open Issues. Delivered to DOE as part of Milestone 4.2	Jerome Soumagne, Quincey Koziol, The HDF Group
2013-09-26	3.0	Minor modifications to NA API / Add checksum section	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 function call (with or without large data arguments) to be forwarded to remote nodes; the network implementation is abstracted so that alternate mechanisms can be implemented and selected, making use of the transport mechanisms natively supported 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.

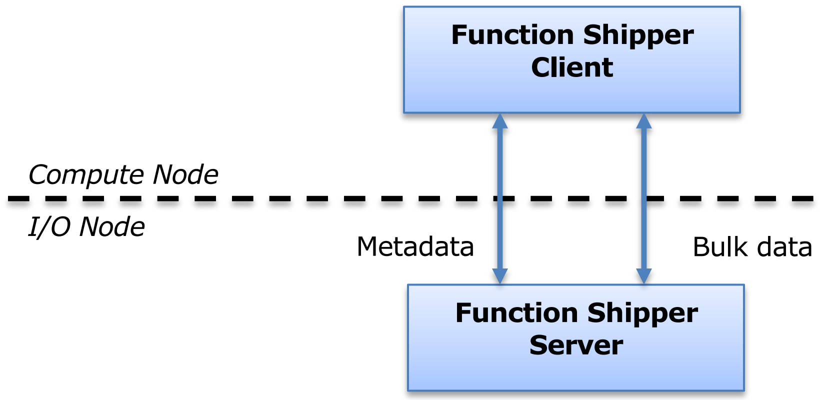

Figure 1. Client/server architecture.

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.

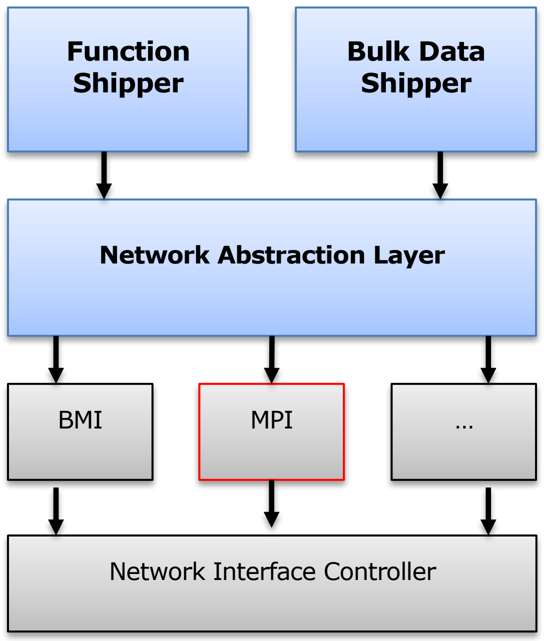

Figure 2. Function shipper interfaces.

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 one particular plugin, one can make use of another plugin to perform bulk data transfers).

In the following sections we only consider an MPI and a BMI plugin as the network abstraction layer plugins that are currently supported. Additional plugins that support native transport protocols and RMA semantics are being added but not fully operational yet.

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 using a non-blocking and 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 of the function shipper, the memory buffer used to receive the response back from the server is also pre-posted by the client.

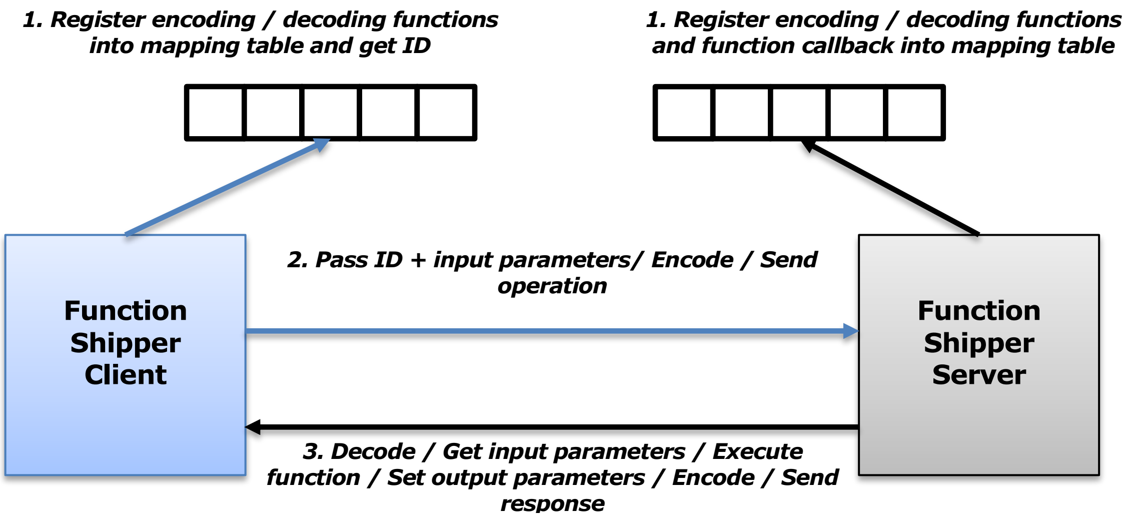
1. Register encoding / decoding functions into mapping table and get ID
1. Register encoding / decoding functions and function callback into mapping table
Function Shipper Client
2. Pass ID + input parameters/ Encode / Send operation
Function Shipper Server
3. Decode / Get input parameters / Execute function / Set output parameters / Encode / Send response

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 sending the response is also non-blocking. 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.

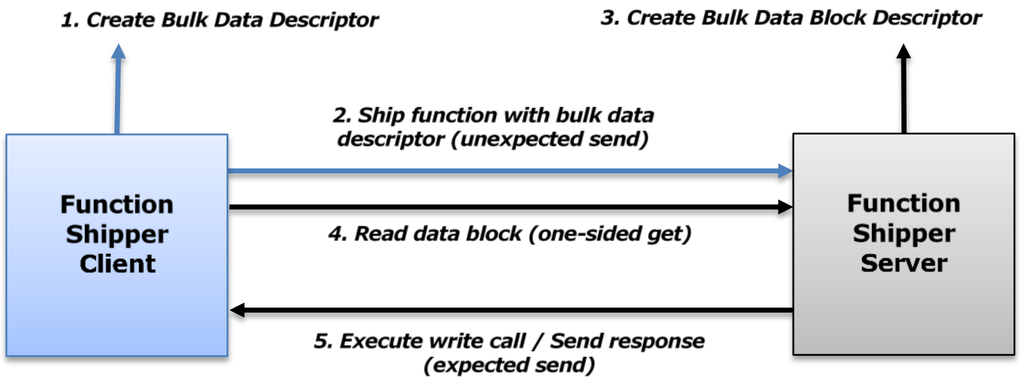
1. Create Bulk Data Descriptor
3. Create Bulk Data Block Descriptor
2. Ship function with bulk data descriptor (unexpected send)
Function Shipper Client
4. Read data block (one-sided get)
Function Shipper Server
5. Execute write call / Send response (expected send)

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 an abstract memory region descriptor 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 must 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 a non-blocking read/get operation on that memory region. The function shipper server then tests 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 a non-blocking write/put operation to the client memory region that has been exposed. The function shipper server then tests the completion of the operation and sends the response (i.e., the result of the *read* call) back to the function shipper client. Memory handles can be freed once the bulk data is successfully received.

In the case of non-contiguous bulk data transfers, non-contiguous regions are exposed and abstracted using the same mechanism, which consists of creating a bulk data descriptor, which is then sent to the server to transfer data to/from its memory. Note that while the memory region exposed by the client is non-contiguous, the local region exposed by the server is always contiguous. Therefore, non-contiguous transfers can also be seen as a gather operation (read) or a scatter operation (write).

Network Plugins

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

The plugins implement 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 these two plugins implement one-sided communication on top of two-sided—the reason being that BMI does not expose RMA semantics through its API and MPI 2 RMA semantics are too restrictive for our use (although tests are being conducted 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). Other plugins that can support 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 modifications realized to the API, as well as the higher level function shipper and bulk data shipper APIs.

Network Abstraction Layer API

Minor modifications to the API: maximum size of expected and unexpected messages has been split; a get_max_tag call has been added so that different maximum tag values can be returned by plugins (which also allows plugins to internally reserve a set of private tag values); a NA_Request_free call has been added to be able to free a request without having to wait for it (useful for cancelation purposes); a NA_Initialize call has been added so that plugins can be selected by passing the corresponding string method.

```
/* Initialize the network abstraction layer */
na_class_t *NA_Initialize(const char *method, const char *port_name,
        na_bool_t listen);

/* Finalize the network abstraction layer */
int NA_Finalize(na_class_t *network_class);

/* Lookup an addr from a peer address/name */
int NA_Addr_lookup(na_class_t *network_class,
        const char *name, na_addr_t *addr);

/* Free the addr from the list of peers */
int NA_Addr_free(na_class_t *network_class,
        na_addr_t addr);

/* Get the maximum size of an expected message */
na_size_t NA_Msg_get_max_expected_size(na_class_t *network_class);

/* Get the maximum size of an unexpected message */
na_size_t NA_Msg_get_max_unexpected_size(na_class_t *network_class);

/* Get the maximum tag value that can be used by NA_Msg routines */
na_tag_t NA_Msg_get_max_tag(na_class_t *network_class);

/* Send an unexpected message to dest */
int NA_Msg_send_unexpected(na_class_t *network_class,
        const void *buf, na_size_t buf_size, na_addr_t dest,
        na_tag_t tag, na_request_t *request, void *op_arg);

/* Receive an unexpected message */
int NA_Msg_recv_unexpected(na_class_t *network_class,
        void *buf, na_size_t buf_size, na_size_t *actual_buf_size,
        na_addr_t *source, na_tag_t *tag, na_request_t *request, void *op_arg);

/* Send an expected message to dest */
int NA_Msg_send(na_class_t *network_class,
        const void *buf, na_size_t buf_size, na_addr_t dest,
        na_tag_t tag, na_request_t *request, void *op_arg);

/* Receive an expected message from source */
int NA_Msg_recv(na_class_t *network_class,
        void *buf, na_size_t buf_size, 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_class_t *network_class,
        void *buf, na_size_t buf_size, unsigned long flags,
        na_mem_handle_t *mem_handle);

/* Register segmented memory for RMA operations */
int NA_Mem_register_segments(na_class_t *network_class,
        na_segment_t *segments, na_size_t segment_count, unsigned long flags,
        na_mem_handle_t *mem_handle);

/* Deregister memory */
int NA_Mem_deregister(na_class_t *network_class,
        na_mem_handle_t mem_handle);

/* Get size required to serialize handle */
na_size_t NA_Mem_handle_get_serialize_size(na_class_t *network_class,
        na_mem_handle_t mem_handle);

/* Serialize memory handle into a buffer */
int NA_Mem_handle_serialize(na_class_t *network_class,
        void *buf, na_size_t buf_size, na_mem_handle_t mem_handle);

/* Deserialize memory handle from buffer */
int NA_Mem_handle_deserialize(na_class_t *network_class,
        na_mem_handle_t *mem_handle, const void *buf, na_size_t buf_size);

/* Free memory handle */
int NA_Mem_handle_free(na_class_t *network_class,
        na_mem_handle_t mem_handle);

/* Put data to remote target */
int NA_Put(na_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_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_class_t *network_class,
        na_request_t request, unsigned int timeout, na_status_t *status);

/* Track completion of RMA operations and make progress */
int NA_Progress(na_class_t *network_class,
        unsigned int timeout, na_status_t *status);
```

```
/* Free a request without waiting for it to complete (request may be active) */
int NA_Request_free(na_class_t *network_class, na_request_t request);
```

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:

```
/* MERCURY_GEN_PROC( struct_type_name, fields ) */
```

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

```
MERCURY_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 hg_proc_function_in_t */
static inline int hg_proc_function_in_t(hg_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 (`hg_string_t`) and bulk data handles (`hg_bulkt_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 `HG_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 HG_Init(na_class_t *network_class);

/* Finalize the function shipper */
int HG_Finalize(void);

/* Indicate whether HG_Init has been called and return associated network class */
int HG_Initialized(bool *flag, na_class_t **network_class);

/* Register a function name and provide a unique ID */
hg_id_t HG_Register(const char *func_name,
        int (*enc_routine)(hg_proc_t proc, void *in_struct),
        int (*dec_routine)(hg_proc_t proc, void *out_struct));

/* Indicate whether HG_Register has been called and return associated ID */
int HG_Registered(const char *func_name, bool *flag, hg_id_t *id);

/* Forward a call to a remote server */
int HG_Forward(na_addr_t addr, hg_id_t id,
        const void *in_struct, void *out_struct, hg_request_t *request);

/* Wait for an operation request to complete */
int HG_Wait(hg_request_t request, unsigned int timeout, hg_status_t *status);

/* Wait for all operations to complete */
int HG_Wait_all(int count, hg_request_t array_of_requests[],
        unsigned int timeout, hg_status_t array_of_statuses[]);
```

Function Shipper handler API (server)

The function shipper handler is only used on the server. The main `HG_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 `HG_Handler_get_input` and `HG_Handler_start_output` in addition to performing the remote operation. Note that this call is non-blocking and corresponding resources are freed when it completes, progress being made during `HG_Handler_process calls.`

The following routines compose the server API:

```
/* Initialize the function shipper handler and select a network protocol */
int HG_Handler_init(na_class_t *network_class);

/* Finalize the function shipper handler */
int HG_Handler_finalize(void);

/* Register a function name and provide a unique ID */
void HG_Handler_register(const char *func_name,
        int (*callback_routine) (hg_handle_t handle),
        int (*dec_routine)(hg_proc_t proc, void *in_struct),
        int (*enc_routine)(hg_proc_t proc, void *out_struct));

/* Get remote addr from handle */
na_addr_t HG_Handler_get_addr(hg_handle_t handle);

/* Get input from handle */
int HG_Handler_get_input_buf(hg_handle_t handle, void **in_buf, size_t
*in_buf_size);

/* Get output from handle */
int HG_Handler_get_output_buf(hg_handle_t handle, void **out_buf, size_t
*out_buf_size);

/* Receive a call from a remote client and process request */
int HG_Handler_process(unsigned int timeout, hg_status_t *status);

/* Send the response back to the remote client and free handle */
int HG_Handler_start_response(hg_handle_t handle, const void *extra_out_buf,
size_t extra_out_buf_size);

/* Wait for a response to complete */
int HG_Handler_wait_response(hg_handle_t handle, unsigned int timeout, hg_status_t
*status);

/* Free the handle (N.B. called in hg_handler_respond) */
int HG_Handler_free(hg_handle_t handle);

/* NB. The following routines are added for convenience */

/* Get input structure from handle (requires registration of decoding function)
 * => HG_Handler_get_input_buf + decode
 */
int HG_Handler_get_input(hg_handle_t handle, void *in_struct);

/* Start sending output structure from handle (requires registration of encoding
function)
 * => HG_Handler_get_output_buf + encode + HG_Handler_start_response
 */
int HG_Handler_start_output(hg_handle_t handle, 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 initiates transfers. The client only uses the first functions (`HG_Bulk_handle_create`, `HG_Bulk_handle_create_segments`, `HG_Bulk_handle_free`, `HG_Bulk_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 when registering non-contiguous memory regions using `HG_Bulk_handle_create_segments`, the memory handle produced may be variable size depending on the network abstraction layer plugin used. If the corresponding serialized memory handle is too large to be sent using the function shipper interface (which makes use of unexpected messaging), a bulk data descriptor of the buffer that contains the serialized handle is automatically created and sent to the server, which can in turn pull that buffer using the bulk data shipper API.

Note also that `HG_Bulk_read_all` and `HG_Bulk_write_all` are provided for convenience and are wrappers built on top of the existing read and write calls.

The following routines compose the bulk data shipper API:

```
/* Initialize the bulk data shipper and select a network protocol */
int HG_Bulk_init(na_class_t *network_class);

/* Finalize */
int HG_Bulk_finalize(void);

/* Indicate whether HG_Bulk_init has been called and return associated network
class */
int HG_Bulk_initialized(bool *flag, na_class_t **network_class);

/* Create bulk data handle from buffer (register memory, etc) */
int HG_Bulk_handle_create(void *buf, size_t buf_size, unsigned long flags,
        hg_bulk_t *handle);

/* Create bulk data handle from arbitrary memory regions */
int HG_Bulk_handle_create_segments(hg_bulk_segment_t *bulk_segments,
        size_t segment_count, unsigned long flags, hg_bulk_t *handle);

/* Free bulk data handle */
int HG_Bulk_handle_free(hg_bulk_t handle);

/* Get data size from handle */
size_t HG_Bulk_handle_get_size(hg_bulk_t handle);

/* Get size required to serialize handle */
size_t HG_Bulk_handle_get_serialize_size(hg_bulk_t handle);

/* Serialize bulk data handle into buf */
int HG_Bulk_handle_serialize(void *buf, size_t buf_size, hg_bulk_t handle);

/* Deserialize bulk data handle from buf */
int HG_Bulk_handle_deserialize(hg_bulk_t *handle, const void *buf, size_t
buf_size);

/* Create bulk data handle from buffer (register memory, etc) */
int HG_Bulk_block_handle_create(void *buf, size_t buf_size, unsigned long flags,
        hg_bulk_block_t *handle);

/* Free block handle */
int HG_Bulk_block_handle_free(hg_bulk_block_t block_handle);

/* Get data size from block handle */
size_t HG_Bulk_block_handle_get_size(hg_bulk_block_t block_handle);

/* Write data */
int HG_Bulk_write(na_addr_t addr, hg_bulk_t bulk_handle, ptrdiff_t bulk_offset,
        hg_bulk_block_t block_handle, ptrdiff_t block_offset, size_t block_size,
        hg_bulk_request_t *bulk_request);

/* Write all the data at the address contained in the bulk handle */
int HG_Bulk_write_all(na_addr_t addr, hg_bulk_t bulk_handle,
        hg_bulk_block_t block_handle, hg_bulk_request_t *bulk_request);
```

```
/* Read data */
int HG_Bulk_read(na_addr_t addr, hg_bulk_t bulk_handle, ptrdiff_t bulk_offset,
        hg_bulk_block_t block_handle, ptrdiff_t block_offset, size_t block_size,
        hg_bulk_request_t *bulk_request);

/* Read all the data from the address contained in the bulk handle */
int HG_Bulk_read_all(na_addr_t addr, hg_bulk_t bulk_handle,
        hg_bulk_block_t block_handle, hg_bulk_request_t *bulk_request);

/* Wait for bulk data operation to complete */
int HG_Bulk_wait(hg_bulk_request_t bulk_request, unsigned int timeout,
        hg_bulk_status_t *status);
```

Abstract checksums and Checksum API

Computing checksums on data encoded and decoded by Mercury has the advantage of guaranteeing to the caller that not only the data has been correctly transmitted over the network but also that *every* parameter encoded has been decoded. Checksums in Mercury are only provided for metadata and it is left to the upper layer to checksum bulk data.

Internal metadata checksumming is implemented by using an abstract checksum object. Currently a CRC64 method has been implemented, but multiple methods can be implemented in the future and dynamically selected. An abstract checksum is attached to every Mercury proc object that encodes or decodes data. The checksum is incrementally updated every time a proc routine is called. Finally once all parameters have been encoded or decoded, the checksum's hash value is retrieved from the abstract checksum object. The hash value is then transmitted along with the metadata to the server or client (in the case of an encoding operation) or used to verify that the data matches the checksum that was previously received (in the case of a decoding operation).

```
/* Initialize the checksum with the specified hash method. */
int hg_checksum_init(const char *hash_method, hg_checksum_t *checksum);

/* Destroy the checksum. */
int hg_checksum_destroy(hg_checksum_t checksum);

/* Reset the checksum. */
int hg_checksum_reset(hg_checksum_t checksum);

/* Get size of checksum. */
size_t hg_checksum_get_size(hg_checksum_t checksum);

/* Get checksum and copy it into buf  (finalize to add padding). */
int hg_checksum_get(hg_checksum_t checksum, void *buf, size_t size, int finalize);

/* Accumulate a partial checksum of the input data. */
int hg_checksum_update(hg_checksum_t checksum, const void *data, size_t size);
```

Open Issues

Bulk data transfers

The bulk data API allows large data (contiguous or non-contiguous) to be efficiently transferred. However, in most cases, executing the RPC call on the server requires all the data to be transferred before it can be actually executed. This introduces two potential issues: the data must fit in the server memory and the user has to pay the cost of the latency introduced by a full RMA transfer.

To prevent these two issues, overlapping transfers and execution by using fine-grained transfers (which our API enables) and pipelining techniques should be encouraged as much as possible in the future.

Risks & Unknowns

The main unknown currently is the final network abstraction layer implementation, which should be addressed in the next milestones so that specific network plugins can be developed, which will implement both unexpected messaging and RMA transfers efficiently. This will be addressed when the testing system is available.