Date: **2014-06-30**	# 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

Revision History

Date	Revision	Notes	Author
March 21, 2013	1.0	• Initial version	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
2014-06-30	4.0	• Update to latest Mercury API – add coresident notes	Jerome Soumagne, The HDF Group
2014-06-30	4.1	• Review and minor edits • Delivered to DOE as part of Milestone 8.5	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 or RPC. 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 are decoded and executed—with the operation's result being sent back to the issuing node. This document describes the implementation of a function shipping framework, also known as Mercury, 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 Milestone 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 easily be transferred across the network.

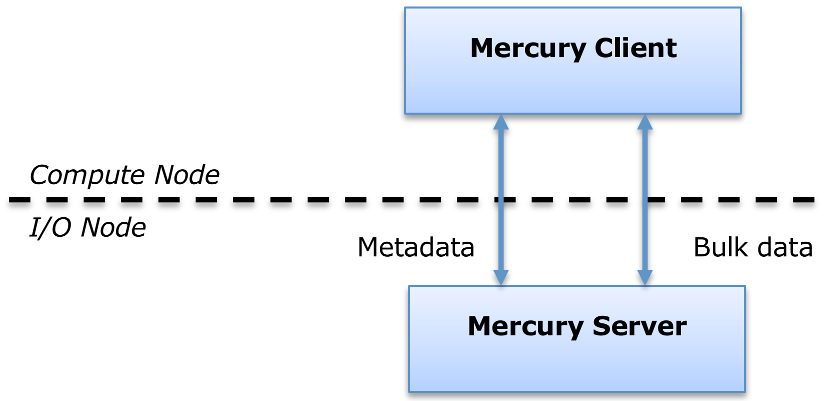

Figure 1. Client/server architecture.

The function shipper client would typically be 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 may connect to other servers, being in turn "clients".

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.

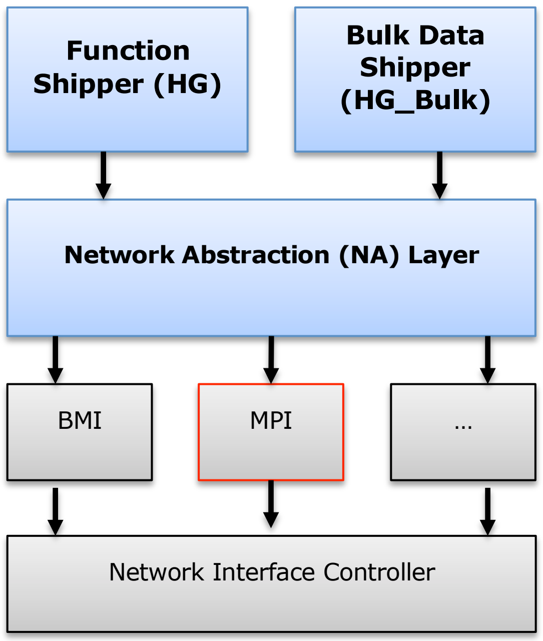
Function Shipper (HG)
Bulk Data Shipper (HG_Bulk)
Network Abstraction (NA) Layer
BMI
MPI
...
Network Interface Controller

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.

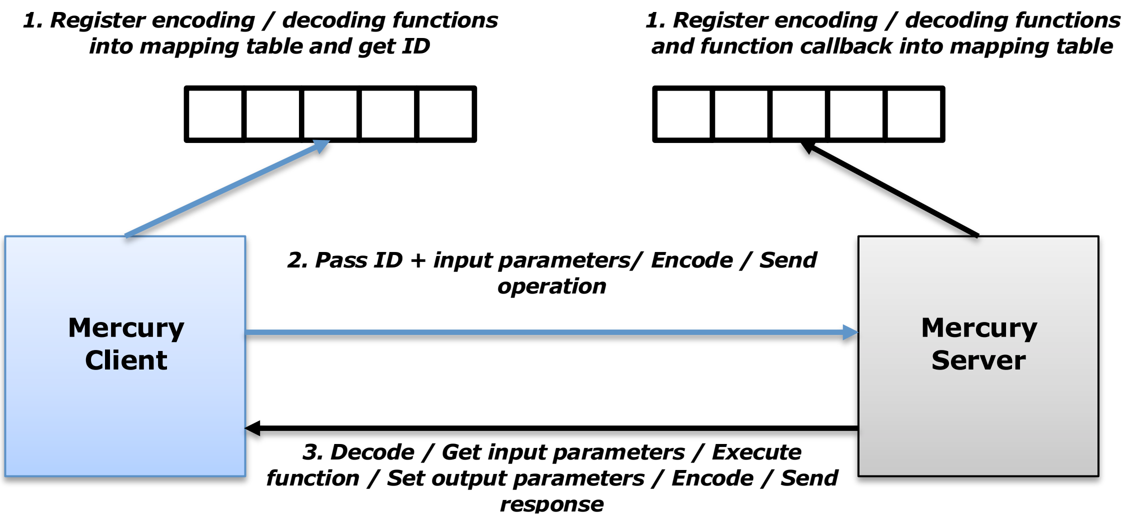
1. Register encoding / decoding functions into mapping table and get ID
1. Register encoding / decoding functions and function callback into mapping table
2. Pass ID + input parameters/ Encode / Send operation
Mercury Client
Mercury Server
3. Decode / Get input parameters / Execute function / Set output 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 for 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 Mercury bulk interface is used.

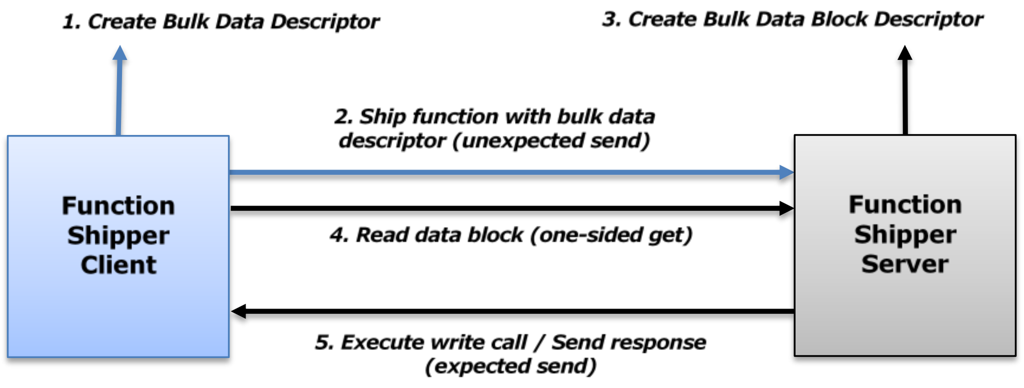
1. Create Bulk Data Descriptor
3. Create Bulk Data Block Descriptor
2. Ship function with bulk data descriptor (unexpected send)
Function Shipper Client
4. Read data block (one-sided get)
Function Shipper Server
5. Execute write call / Send response (expected send)

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 Mercury client exposes an abstract memory region descriptor to the 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 arguments. When the server decodes the input arguments 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. 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 generally not supported (although a solution has been implemented by Cray recently), 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

The Network Abstraction (NA) layer uses a callback model, which can be considered similar to an event-based model. As opposed to the original request-based model introduced in Milestone 2.4, the callback model removes the "wait on request" limitation, which requires waiting on a particular request object, hence forcing continuous polling most of the time. In this callback model, non-blocking calls such as lookup, send_unexpected, put, etc, take a user callback function pointer argument, as well as a pointer to user data that is passed to the user callback function. Additionally these calls use a NA context. Progress on communication is made using NA_Progress on a specific context. When the operation completes, the user callback is pushed into a completion queue that is specific to the context. The user callback gets executed and popped from the completion after NA_Trigger is called.

Other NA API calls are presented and documented below:

```
/**
 * Initialize the network abstraction layer.
 *
 * \param info_string [IN]      host address with port number (e.g.,
 *                              "tcp://localhost:3344" or
 *                              "bmi+tcp://localhost:3344")
 * \param listen [IN]           listen for incoming connections
 *
 * \return Pointer to NA class or NULL in case of failure
 */
NA_EXPORT na_class_t *
NA_Initialize(
        const char *info_string,
        na_bool_t   listen
        ) NA_WARN_UNUSED_RESULT;

/**
 * Finalize the network abstraction layer.
 *
 * \param na_class [IN]         pointer to NA class
 *
 * \return NA_SUCCESS or corresponding NA error code
 */
NA_EXPORT na_return_t
NA_Finalize(
        na_class_t *na_class
        );

/**
 * Create a new context.
 *
 * \param na_class [IN]         pointer to NA class
 *
 * \return Pointer to NA context or NULL in case of failure
 */
NA_EXPORT na_context_t *
NA_Context_create(
        na_class_t *na_class
        );

/**
 * Destroy a context created by using NA_Context_create().
 *
 * \param na_class [IN]         pointer to NA class
 * \param context [IN]          pointer to context of execution
 *
 * \return NA_SUCCESS or corresponding NA error code
 */
NA_EXPORT na_return_t
NA_Context_destroy(
        na_class_t   *na_class,
        na_context_t *context
        );

/**
 * Lookup an addr from a peer address/name. Addresses need to be
 * freed by calling NA_Addr_free. Callback will be called with pointer to
 * na_addr_t that contains addr ::na_cb_info_lookup
 *
 * \param na_class [IN]         pointer to NA class
 * \param context [IN]          pointer to context of execution
 * \param callback [IN]         pointer to function callback
 * \param arg [IN]              pointer to data passed to callback
 * \param name [IN]             lookup name
 * \param op_id [OUT]           pointer to returned operation ID
 *
 * \return NA_SUCCESS or corresponding NA error code
 */
NA_EXPORT na_return_t
NA_Addr_lookup(
        na_class_t   *na_class,
        na_context_t *context,
        na_cb_t       callback,
        void         *arg,
        const char   *name,
        na_op_id_t   *op_id
        );

/**
 * Free the addr from the list of peers.
 *
 * \param na_class [IN]         pointer to NA class
 * \param addr [IN]             abstract address
 *
 * \return NA_SUCCESS or corresponding NA error code
 */
NA_EXPORT na_return_t
NA_Addr_free(
        na_class_t *na_class,
        na_addr_t   addr
```

```c
        );

/**
 * Access self address.
 *
 * \param na_class [IN]         pointer to NA class
 * \param addr [OUT]            pointer to abstract address
 *
 * \return NA_SUCCESS or corresponding NA error code
 */
NA_EXPORT na_return_t
NA_Addr_self(
        na_class_t *na_class,
        na_addr_t  *addr
        );

/**
 * Duplicate an existing NA abstract address. The duplicated address can be
 * stored for later use and the origin address be freed safely. The duplicated
 * address must be freed with NA_Addr_free.
 *
 * \param na_class [IN]         pointer to NA class
 * \param addr [IN]             abstract address
 * \param new_addr [OUT]        pointer to abstract address
 *
 * \return NA_SUCCESS or corresponding NA error code
 */
NA_EXPORT na_return_t
NA_Addr_dup(
        na_class_t *na_class,
        na_addr_t   addr,
        na_addr_t  *new_addr
        );

/**
 * Test whether address is self or not.
 *
 * \param na_class [IN]         pointer to NA class
 * \param addr [IN]             abstract address
 *
 * \return NA_TRUE if self or NA_FALSE if not
 */
NA_EXPORT na_bool_t
NA_Addr_is_self(
        na_class_t *na_class,
        na_addr_t   addr
        );

/**
 * Convert an addr to a string (returned string includes the terminating
 * null byte '\0').
 *
 * \param na_class [IN]         pointer to NA class
 * \param buf [IN/OUT]          pointer to destination buffer
 * \param buf_size [IN]         buffer size (max string length is defined
 *                              by NA_MAX_ADDR_LEN)
 * \param addr [IN]             abstract address
 *
 * \return NA_SUCCESS or corresponding NA error code
 */
NA_EXPORT na_return_t
NA_Addr_to_string(
        na_class_t *na_class,
        char       *buf,
        na_size_t   buf_size,
        na_addr_t   addr
        );

/**
 * Get the maximum size of messages supported by expected send/recv.
 * Small message size that may differ from the unexpected message size.
 *
 * \param na_class [IN]         pointer to NA class
 *
 * \return Non-negative value
 */
NA_EXPORT na_size_t
NA_Msg_get_max_expected_size(
        na_class_t *na_class
        ) NA_WARN_UNUSED_RESULT;

/**
 * Get the maximum size of messages supported by unexpected send/recv.
 * Small message size.
 *
 * \param na_class [IN]         pointer to NA class
 *
 * \return Non-negative value
 */
NA_EXPORT na_size_t
NA_Msg_get_max_unexpected_size(
        na_class_t *na_class
        ) NA_WARN_UNUSED_RESULT;

/**
 * Get the maximum tag value that can be used by send/recv.
 * (both expected and unexpected)
 *
 * \param na_class [IN]         pointer to NA class
 *
 * \return Non-negative value
 */
NA_EXPORT na_tag_t
NA_Msg_get_max_tag(
        na_class_t *na_class
        ) NA_WARN_UNUSED_RESULT;

/**
 * Send an unexpected message to dest.
 * Unexpected sends do not require a matching receive to complete.
 * Note also that unexpected messages do not require an unexpected receive to
 * be posted at the destination before sending the message and the destination
 * is allowed to drop the message without notification.
 *
 * \param na_class [IN]         pointer to NA class
 * \param context [IN]          pointer to context of execution
 * \param callback [IN]         pointer to function callback
 * \param arg [IN]              pointer to data passed to callback
 * \param buf [IN]              pointer to send buffer
 * \param buf_size [IN]         buffer size
 * \param dest [IN]             abstract address of destination
 * \param tag [IN]              tag attached to message
 * \param op_id [OUT]           pointer to returned operation ID
 *
 * \return NA_SUCCESS or corresponding NA error code
 */
NA_EXPORT na_return_t
NA_Msg_send_unexpected(
        na_class_t   *na_class,
        na_context_t *context,
        na_cb_t       callback,
        void         *arg,
        const void   *buf,
        na_size_t     buf_size,
        na_addr_t     dest,
        na_tag_t      tag,
        na_op_id_t   *op_id
        );

/**
 * Receive an unexpected message.
 * Unexpected receives may wait on ANY_TAG and ANY_SOURCE depending on the
 * implementation.
 *
 * \param na_class [IN]         pointer to NA class
 * \param context [IN]          pointer to context of execution
 * \param callback [IN]         pointer to function callback
 * \param arg [IN]              pointer to data passed to callback
 * \param buf [IN]              pointer to send buffer
 * \param buf_size [IN]         buffer size
 * \param op_id [OUT]           pointer to returned operation ID
 *
 * \return NA_SUCCESS or corresponding NA error code
 */
NA_EXPORT na_return_t
NA_Msg_recv_unexpected(
        na_class_t   *na_class,
        na_context_t *context,
        na_cb_t       callback,
        void         *arg,
        void         *buf,
        na_size_t     buf_size,
        na_op_id_t   *op_id
        );

/**
 * Send an expected message to dest. Note that expected messages require
 * an expected receive to be posted at the destination before sending the
 * message, otherwise the destination is allowed to drop the message without
 * notification.
 *
 * \param na_class [IN]         pointer to NA class
 * \param context [IN]          pointer to context of execution
 * \param callback [IN]         pointer to function callback
 * \param arg [IN]              pointer to data passed to callback
 * \param buf [IN]              pointer to send buffer
 * \param buf_size [IN]         buffer size
 * \param dest [IN]             abstract address of destination
 * \param tag [IN]              tag attached to message
 * \param op_id [OUT]           pointer to returned operation ID
 *
 * \return NA_SUCCESS or corresponding NA error code
 */
NA_EXPORT na_return_t
NA_Msg_send_expected(
        na_class_t   *na_class,
        na_context_t *context,
        na_cb_t       callback,
        void         *arg,
        const void   *buf,
        na_size_t     buf_size,
        na_addr_t     dest,
        na_tag_t      tag,
        na_op_id_t   *op_id
        );

/**
 * Receive an expected message from source.
 *
 * \param na_class [IN]         pointer to NA class
 * \param context [IN]          pointer to context of execution
 * \param callback [IN]         pointer to function callback
 * \param arg [IN]              pointer to data passed to callback
 * \param buf [IN]              pointer to receive buffer
 * \param buf_size [IN]         buffer size
 * \param source [IN]           abstract address of source
 * \param tag [IN]              matching tag used to receive message
 * \param op_id [OUT]           pointer to returned operation ID
 *
 * \return NA_SUCCESS or corresponding NA error code
 */
NA_EXPORT na_return_t
NA_Msg_recv_expected(
        na_class_t   *na_class,
        na_context_t *context,
        na_cb_t       callback,
        void         *arg,
        void         *buf,
        na_size_t     buf_size,
        na_addr_t     source,
        na_tag_t      tag,
        na_op_id_t   *op_id
        );
```

```
/**
 * Create memory handle for RMA operations.
 * For non-contiguous memory, use NA_Mem_handle_create_segments instead.
 *
 * Note to plugin developers: NA_Mem_handle_create may be called multiple times
 * on the same memory region.
 *
 * \param na_class [IN]         pointer to NA class
 * \param buf [IN]              pointer to buffer that needs to be registered
 * \param buf_size [IN]         buffer size
 * \param flags [IN]            permission flag:
 *                                - NA_MEM_READWRITE
 *                                - NA_MEM_READ_ONLY
 * \param mem_handle [OUT]      pointer to returned abstract memory handle
 *
 * \return NA_SUCCESS or corresponding NA error code
 */
NA_EXPORT na_return_t
NA_Mem_handle_create(
        na_class_t      *na_class,
        void            *buf,
        na_size_t        buf_size,
        unsigned long    flags,
        na_mem_handle_t *mem_handle
        );

/**
 * Create memory handle for RMA operations.
 * Create_segments can be used to register fragmented pieces and get
 * a single memory handle.
 * Implemented only if the network transport or hardware supports it.
 *
 * \param na_class [IN]         pointer to NA class
 * \param segments [IN]         pointer to array of segments composed of:
 *                                - address of the segment that needs to be
 *                                  registered
 *                                - size of the segment in bytes
 * \param segment_count [IN]    segment count
 * \param flags [IN]            permission flag:
 *                                - NA_MEM_READWRITE
 *                                - NA_MEM_READ_ONLY
 * \param mem_handle [OUT]      pointer to returned abstract memory handle
 *
 * \return NA_SUCCESS or corresponding NA error code
 */
NA_EXPORT na_return_t
NA_Mem_handle_create_segments(
        na_class_t        *na_class,
        struct na_segment *segments,
        na_size_t          segment_count,
        unsigned long      flags,
        na_mem_handle_t   *mem_handle
        );

/**
 * Free memory handle.
 *
 * \param na_class [IN]         pointer to NA class
 * \param mem_handle [IN]       abstract memory handle
 *
 * \return NA_SUCCESS or corresponding NA error code
 */
NA_EXPORT na_return_t
NA_Mem_handle_free(
        na_class_t      *na_class,
        na_mem_handle_t  mem_handle
        );

/**
 * Register memory for RMA operations.
 * Memory pieces must be registered before one-sided transfers can be
 * initiated.
 *
 * \param na_class [IN]         pointer to NA class
 * \param mem_handle [IN]       pointer to abstract memory handle
 *
 * \return NA_SUCCESS or corresponding NA error code
 */
NA_EXPORT na_return_t
NA_Mem_register(
        na_class_t      *na_class,
        na_mem_handle_t  mem_handle
        );

/**
 * Unregister memory.
 *
 * \param na_class [IN]         pointer to NA class
 * \param mem_handle [IN]       abstract memory handle
 *
 * \return NA_SUCCESS or corresponding NA error code
 */
NA_EXPORT na_return_t
NA_Mem_deregister(
        na_class_t      *na_class,
        na_mem_handle_t  mem_handle
        );

/**
 * Get size required to serialize handle.
 *
 * \param na_class [IN]         pointer to NA class
 * \param mem_handle [IN]       abstract memory handle
 *
 * \return Non-negative value
 */
NA_EXPORT na_size_t
NA_Mem_handle_get_serialize_size(
        na_class_t      *na_class,
        na_mem_handle_t  mem_handle
        ) NA_WARN_UNUSED_RESULT;
```

```
/**
 * Serialize memory handle into a buffer.
 * One-sided transfers require prior exchange of memory handles between
 * peers, serialization callbacks can be used to "pack" a memory handle and
 * send it across the network.
 * NB. Memory handles can be variable size, therefore the space required
 * to serialize a handle into a buffer can be obtained using
 * NA_Mem_handle_get_serialize_size.
 *
 * \param na_class [IN]         pointer to NA class
 * \param buf [IN/OUT]          pointer to buffer used for serialization
 * \param buf_size [IN]         buffer size
 * \param mem_handle [IN]       abstract memory handle
 *
 * \return NA_SUCCESS or corresponding NA error code
 */
NA_EXPORT na_return_t
NA_Mem_handle_serialize(
        na_class_t      *na_class,
        void            *buf,
        na_size_t        buf_size,
        na_mem_handle_t  mem_handle
        );

/**
 * Deserialize memory handle from buffer.
 *
 * \param na_class [IN]         pointer to NA class
 * \param mem_handle [OUT]      pointer to abstract memory handle
 * \param buf [IN]              pointer to buffer used for deserialization
 * \param buf_size [IN]         buffer size
 *
 * \return NA_SUCCESS or corresponding NA error code
 */
NA_EXPORT na_return_t
NA_Mem_handle_deserialize(
        na_class_t      *na_class,
        na_mem_handle_t *mem_handle,
        const void      *buf,
        na_size_t        buf_size
        );

/**
 * Put data to remote target.
 * Initiate a put or get to/from the registered memory regions with the
 * given offset/size.
 * NB. Memory must be registered and handles exchanged between peers.
 *
 * \param na_class [IN]          pointer to NA class
 * \param context [IN]           pointer to context of execution
 * \param callback [IN]          pointer to function callback
 * \param arg [IN]               pointer to data passed to callback
 * \param local_mem_handle [IN]  abstract local memory handle
 * \param local_offset [IN]      local offset
 * \param remote_mem_handle [IN] abstract remote memory handle
 * \param remote_offset [IN]     remote offset
 * \param data_size [IN]         size of data that needs to be transferred
 * \param remote_addr [IN]       abstract address of remote destination
 * \param op_id [OUT]            pointer to returned operation ID
 *
 * \return NA_SUCCESS or corresponding NA error code
 */
NA_EXPORT na_return_t
NA_Put(
        na_class_t      *na_class,
        na_context_t    *context,
        na_cb_t          callback,
        void            *arg,
        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        data_size,
        na_addr_t        remote_addr,
        na_op_id_t      *op_id
        );

/**
 * Get data from remote target.
 *
 * \param na_class [IN]          pointer to NA class
 * \param context [IN]           pointer to context of execution
 * \param callback [IN]          pointer to function callback
 * \param arg [IN]               pointer to data passed to callback
 * \param local_mem_handle [IN]  abstract local memory handle
 * \param local_offset [IN]      local offset
 * \param remote_mem_handle [IN] abstract remote memory handle
 * \param remote_offset [IN]     remote offset
 * \param data_size [IN]         size of data that needs to be transferred
 * \param remote_addr [IN]       abstract address of remote source
 * \param op_id [OUT]            pointer to returned operation ID
 *
 * \return NA_SUCCESS or corresponding NA error code
 */
NA_EXPORT na_return_t
NA_Get(
        na_class_t      *na_class,
        na_context_t    *context,
        na_cb_t          callback,
        void            *arg,
        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        data_size,
        na_addr_t        remote_addr,
        na_op_id_t      *op_id
        );

/**
 * Try to progress communication for at most timeout until timeout reached or
 * any completion has occurred.
 * Progress should not be considered as wait, in the sense that it cannot be
 * assumed that completion of a specific operation will occur only when
 * progress is called.
 *
 * \param na_class [IN]         pointer to NA class
 * \param context [IN]          pointer to context of execution
 * \param timeout [IN]          timeout (in milliseconds)
 *
 * \return NA_SUCCESS if any completion has occurred / NA error code otherwise
 */
NA_EXPORT na_return_t
NA_Progress(
        na_class_t   *na_class,
        na_context_t *context,
        unsigned int  timeout
        );

/**
 * Execute at most max_count callbacks. If timeout is non-zero, wait up to
 * timeout before returning. Function can return when at least one or more
 * callbacks are triggered (at most max_count).
 *
 * \param context [IN]          pointer to context of execution
 * \param timeout [IN]          timeout (in milliseconds)
 * \param max_count [IN]        maximum number of callbacks triggered
 * \param actual_count [IN]     actual number of callbacks triggered
 *
 * \return NA_SUCCESS or corresponding NA error code
 */
NA_EXPORT na_return_t
NA_Trigger(
        na_context_t *context,
        unsigned int  timeout,
        unsigned int  max_count,
        unsigned int *actual_count
        );

/**
 * Cancel an ongoing operation.
 *
 * \param na_class [IN]         pointer to NA class
 * \param context [IN]          pointer to context of execution
 * \param op_id [IN]            operation ID
 *
 * \return NA_SUCCESS or corresponding NA error code
 */
NA_EXPORT na_return_t
NA_Cancel(
        na_class_t   *na_class,
        na_context_t *context,
        na_op_id_t    op_id
        );
```

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_bulk_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.

Mercury 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. If the address passed is "self", the call will not be sent but instead, will be executed locally.

The following routines compose the client API:

```
/**
 * Initialize the Mercury layer.
 * Calling HG_Init also calls HG_Bulk_init with the same NA class if
 * HG_Bulk_init has not been called before, this allows users to
 * eventually initialize the bulk interface with a different NA class.
 *
 * \param na_class [IN]    pointer to network class
 *
 * \return HG_SUCCESS or corresponding HG error code
 */
HG_EXPORT hg_return_t
HG_Init(na_class_t *na_class);

/**
 * Finalize the Mercury layer.
 *
 * \return HG_SUCCESS or corresponding HG error code
 */
HG_EXPORT hg_return_t
HG_Finalize(void);

/**
 * Register a function name that can be sent using the RPC layer.
 *
 * \param func_name [IN]        unique name associated to function
 * \param in_proc_cb [IN]       pointer to input proc routine
 * \param out_proc_cb [IN]      pointer to output proc routine
 * \param rpc_cb [IN]           RPC callback (may only be defined in server code)
 *
 *
 * \return unique ID associated to the registered function
 */
HG_EXPORT hg_id_t
HG_Register(const char *func_name, hg_proc_cb_t in_proc_cb,
        hg_proc_cb_t out_proc_cb, hg_rpc_cb_t rpc_cb);

/**
 * Forward a call to a remote server.
 * Request must be freed using HG_Request_free.
 *
 * \param addr [IN]             abstract network address of destination
 * \param id [IN]               registered function ID
 * \param in_struct [IN]        pointer to input structure
 * \param out_struct [OUT]      pointer to output structure
 * \param request [OUT]         pointer to RPC request
 *
 * \return HG_SUCCESS or corresponding HG error code
 */
HG_EXPORT hg_return_t
HG_Forward(na_addr_t addr, hg_id_t id,
        void *in_struct, void *out_struct, hg_request_t *request);

/**
 * Wait for an operation request to complete.
 * Once the request has completed, request must be freed using HG_Request_free.
 *
 * \param request [IN]          RPC request
 * \param timeout [IN]          timeout (in milliseconds)
 * \param status [OUT]          pointer to returned status
 *
 * \return HG_SUCCESS or corresponding HG error code
 */
HG_EXPORT hg_return_t
HG_Wait(hg_request_t request, unsigned int timeout, hg_status_t *status);

/**
 * Wait for all operations in array_of_requests to complete.
 *
 * \param count [IN]               number of RPC requests
 * \param array_of_requests [IN]   arrays of RPC requests
 * \param timeout [IN]             timeout (in milliseconds)
 * \param array_of_statuses [OUT] array of statuses
 *
 * \return HG_SUCCESS or corresponding HG error code
 */
HG_EXPORT hg_return_t
HG_Wait_all(int count, hg_request_t array_of_requests[],
        unsigned int timeout, hg_status_t array_of_statuses[]);

/**
 * Free request and resources allocated when decoding the output.
 * User must get output parameters contained in the output structure
 * before calling HG_Request_free.
 *
 * \param request [IN]          RPC request
 *
 * \return HG_SUCCESS or corresponding HG error code
 */
HG_EXPORT hg_return_t
HG_Request_free(hg_request_t request);
```

Mercury 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:

```
/**
 * Try timeout ms to process RPC requests.
 *
 * \param timeout [IN]          timeout (in milliseconds)
 * \param status [OUT]          pointer to status object
 *
 * \return HG_SUCCESS or corresponding HG error code
 */
HG_EXPORT hg_return_t
HG_Handler_process(unsigned int timeout, hg_status_t *status);

/**
 * Get abstract network address of remote caller from RPC handle.
 * The address gets freed when HG_Handler_free is called. Users
 * must call NA_Addr_dup to be able to safely re-use the address.
 *
 * \param handle [IN]           abstract RPC handle
 *
 * \return Abstract network address
 */
HG_EXPORT na_addr_t
HG_Handler_get_addr(hg_handle_t handle);

/**
 * Get input from handle (requires registration of input proc to deserialize
 * parameters).
 * This is equivalent to:
 *   - HG_Handler_get_input_buf
 *   - Call hg_proc to deserialize parameters
 *
 * \param handle [IN]           abstract RPC handle
 * \param in_struct [OUT]       pointer to input structure that will be
 *                              filled with deserialized input parameters of
 *                              RPC call.
 *
 * \return HG_SUCCESS or corresponding HG error code
 */
HG_EXPORT hg_return_t
HG_Handler_get_input(hg_handle_t handle, void *in_struct);

/**
 * Free input members allocated during deserialization operation.
 */
HG_EXPORT hg_return_t
HG_Handler_free_input(hg_handle_t handle, void *in_struct);

/**
 * Start sending output from handle (requires registration of output proc to
 * serialize parameters)
 * This is equivalent to:
 *   - HG_Handler_get_output_buf
 *   - Call hg_proc to serialize parameters
 *   - HG_Handler_start_response
 *
 * \param handle [IN]           abstract RPC handle
 * \param out_struct [IN]       pointer to output structure that has been
 *                              filled with output parameters and which will
 *                              be serialized into a buffer. This buffer is then
 *                              sent using a non-blocking expected send.
 *
 * \return HG_SUCCESS or corresponding HG error code
 */
```

```
HG_EXPORT hg_return_t
HG_Handler_start_output(hg_handle_t handle, void *out_struct);

/**
 * Release resources allocated for handling the RPC.
 *
 * \param handle [IN]           abstract RPC handle
 *
 * \return HG_SUCCESS or corresponding HG error code
 */
HG_EXPORT hg_return_t
HG_Handler_free(hg_handle_t handle);
```

Mercury bulk API

The bulk data 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_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`, 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 if the address passed to `HG_Bulk_transfer` is "self", data will be copied locally, from one memory region to the other. To avoid copy, one can use `HG_Bulk_handle_access` directly on the bulk data descriptor and access data pointers transparently.

The following routines compose the bulk data shipper API:

```
/**
 * Initialize the Mercury bulk layer.
 * The NA class can be different from the one used for the RPC interface.
 *
 * \param na_class [IN]    pointer to network class
 *
 * \return HG_SUCCESS or corresponding HG error code
 */
HG_EXPORT hg_return_t
HG_Bulk_init(na_class_t *na_class);

/**
 * Finalize the Mercury bulk layer.
 *
 * \return HG_SUCCESS or corresponding HG error code
 */
HG_EXPORT hg_return_t
HG_Bulk_finalize(void);

/**
 * Create abstract bulk handle from specified memory segments.
 * Note.
 * If NULL is passed to buf_ptrs, i.e.,
 *   HG_Bulk_handle_create(count, NULL, buf_sizes, flags, &handle)
 * memory for the missing buf_ptrs array will be internally allocated.
 * Memory allocated is then freed when HG_Bulk_handle_free is called.
 *
 * \param count [IN]            number of segments
 * \param buf_ptrs [IN]         array of pointers
 * \param buf_sizes [IN]        array of sizes
 * \param flags [IN]            permission flag:
 *                                - HG_BULK_READWRITE
 *                                - HG_BULK_READ_ONLY
 *                                - HG_BULK_WRITE_ONLY
 * \param handle [OUT]          pointer to returned abstract bulk handle
 *
 * \return HG_SUCCESS or corresponding HG error code
 */
HG_EXPORT hg_return_t
HG_Bulk_handle_create(size_t count, void **buf_ptrs, const size_t *buf_sizes,
        unsigned long flags, hg_bulk_t *handle);

/**
 * Free bulk handle.
 *
 * \param handle [IN/OUT]       abstract bulk handle
 *
 * \return HG_SUCCESS or corresponding HG error code
 */
HG_EXPORT hg_return_t
HG_Bulk_handle_free(hg_bulk_t handle);

/**
 * Access bulk handle to retrieve memory segments abstracted by handle.
 * When using mercury in coresident mode (i.e., when addr passed is self addr),
 * it is possible to avoid copy of bulk data by accessing pointers
 * from an existing bulk handle directly.
 *
 * \param handle [IN]            abstract bulk handle
 * \param offset [IN]            bulk offset
 * \param size [IN]              bulk size
 * \param flags [IN]             permission flag:
 *                                 - HG_BULK_READWRITE
 *                                 - HG_BULK_READ_ONLY
 * \param max_count [IN]         maximum number of segments to be returned
 * \param buf_ptrs [IN/OUT]      array of buffer pointers
 * \param buf_sizes [IN/OUT]     array of buffer sizes
 * \param actual_count [OUT]     actual number of segments returned
 *
 * \return HG_SUCCESS or corresponding HG error code
 */
HG_EXPORT hg_return_t
HG_Bulk_handle_access(hg_bulk_t handle, size_t offset, size_t size,
        unsigned long flags, unsigned int max_count, void **buf_ptrs,
        size_t *buf_sizes, unsigned int *actual_count);

/**
 * Get total size of data abstracted by bulk handle.
 *
 * \param handle [IN]           abstract bulk handle
 *
 * \return Non-negative value
 */
HG_EXPORT size_t
HG_Bulk_handle_get_size(hg_bulk_t handle);

/**
 * Get total number of segments abstracted by bulk handle.
 *
 * \param handle [IN]           abstract bulk handle
 *
 * \return Non-negative value
 */
HG_EXPORT size_t
HG_Bulk_handle_get_segment_count(hg_bulk_t handle);

/**
 * Get size required to serialize bulk handle.
 *
 * \param handle [IN]           abstract bulk handle
 *
 * \return Non-negative value
 */
HG_EXPORT size_t
HG_Bulk_handle_get_serialize_size(hg_bulk_t handle);

/**
 * Serialize bulk handle into a buffer.
 *
 * \param buf [IN/OUT]          pointer to buffer
 * \param buf_size [IN]         buffer size
 * \param handle [IN]           abstract bulk handle
 *
 * \return HG_SUCCESS or corresponding HG error code
 */
HG_EXPORT hg_return_t
HG_Bulk_handle_serialize(void *buf, size_t buf_size, hg_bulk_t handle);

/**
 * Deserialize bulk handle from a buffer.
 *
 * \param handle [OUT]          abstract bulk handle
 * \param buf [IN]              pointer to buffer
 * \param buf_size [IN]         buffer size
 *
 * \return HG_SUCCESS or corresponding HG error code
 */
HG_EXPORT hg_return_t
HG_Bulk_handle_deserialize(hg_bulk_t *handle, const void *buf, size_t buf_size);

/**
 * Transfer data to/from origin using abstract bulk handles.
 *
 * \param op [IN]               transfer operation:
```

```
 *                                  - HG_BULK_PUSH
 *                                  - HG_BULK_PULL
 * \param origin_addr [IN]      abstract NA address of origin
 * \param origin_handle [IN]    abstract bulk handle
 * \param origin_offset [IN]    offset
 * \param local_handle [IN]     abstract bulk handle
 * \param local_offset [IN]     offset
 * \param size [IN]             size of data to be transferred
 * \param request [OUT]         pointer to returned bulk request
 *
 * \return HG_SUCCESS or corresponding HG error code
 */
HG_EXPORT hg_return_t
HG_Bulk_transfer(hg_bulk_op_t op, na_addr_t origin_addr, hg_bulk_t origin_handle,
        size_t origin_offset, hg_bulk_t local_handle, size_t local_offset,
        size_t size, hg_bulk_request_t *request);

/**
 * Wait for a bulk operation request to complete.
 *
 * \param request [IN]          bulk request
 * \param timeout [IN]          timeout (in milliseconds)
 * \param status [OUT]          pointer to returned status
 *
 * \return HG_SUCCESS or corresponding HG error code
 */
HG_EXPORT hg_return_t
HG_Bulk_wait(hg_bulk_request_t request, unsigned int timeout,
        hg_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 has the data 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 caller 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.

Callback model

While the network abstraction layer API has been switched to make use of a callback model, the main Mercury API should be switched to that model in order to make progress on completion of RPC calls and global communication more efficiently, using separate threads for example (in a particular context).

Risks & Unknowns

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