Date: **December 13, 2012**	**FUNCTION SHIPPING DESIGN & FRAMEWORK DEMONSTRATION 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	Author
2012-12-13	1.0 Draft for Review	Jerome Soumagne, HDF Group Quincey Koziol, HDF Group

RFC: Function Shipping for Exascale I/O Stack

Using HDF5 on exascale systems will not 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 well-known method called function shipping. Making use of this method, I/O calls issued from the computes 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 defines a function shipping framework that allows I/O calls to be remotely executed in a scalable and asynchronous manner. This interface is designed to be as generic as possible to allow any I/O function call to be forwarded to the I/O nodes. The network implementation is abstracted so that alternate mechanisms can be implemented and selected, making use of the most efficient transport mechanism available on the system.

1 Introduction

Remote procedure call (RPC) (Birrell & Nelson, 1984) is a technique that has been used in many projects to distribute and execute tasks onto remote systems. As opposed to RPC frameworks such as CORBA (OMG, 2012) or Protocol Buffers (Hosting, 2012) the framework that we propose to develop must be specific and optimized to the forwarding of I/O operations. These operations generally fall into two categories: metadata operations, which imply low-latency transfers of small amounts of data; and bulk data operations, which imply high-bandwidth transfers of large amounts of data. The function shipping framework that we propose to develop must be particularly efficient with this last category—which is also the reason why conventional RPC frameworks cannot be directly re-used as they have not been designed for HPC and more importantly to forward operations on large amounts of data (they only make use of standard communication protocols such as TCP, which cannot provide the required performance).

The I/O Forwarding Scalability Layer (IOFSL) (Ali, et al., 2009) is an existing project that has been designed towards that goal, defining a comprehensive set of I/O forwarding calls in the ZOIDFS API, which in turn get mapped onto file system specific I/O operations. Built on top of the BMI (Carns, Ligon, Ross, & Wyckoff, 2005) network transport layer, the interface allows MPI-2 limitations such as fault tolerance, dynamic connection and handling of unexpected messages to be bypassed. However the IOFSL framework is tied to a well-defined and static set of functions that are not easily extendable, which is a limitation for our approach. Moreover the BMI implementation only supports portals 3, which was used on Cray XT systems featuring Seastar interconnects, now replaced by Gemini/Aries interconnects. We therefore propose to keep the same architecture as IOFSL but enhance it in a more generic and transport independent way, which will allow multiple transport plugins to be developed without requiring modification of the framework. We also propose to add new capabilities dedicated to the transfer of bulk data and the forwarding of generic function calls, all of which will be abstracted and independent of the network type that is to be used.

2 Architecture

The approach follows a client/server architecture: the IOFSL server runs on a dedicated I/O node and the client, integrated into the HDF5 library, runs on a compute node. Each compute node may forward I/O calls to different I/O nodes.

2.1 Asynchronous Operations

Every I/O operation issued by HDF5 (referred to as an *I/O request*) is asynchronously shipped to the IOFSL server and asynchronously executed (referred to as an *I/O task*). An IOFSL server may therefore use a pool of threads so that no bottleneck is created by the reception/execution of an I/O request as multiple operations from multiple compute nodes may reach the same I/O node at the same time. This mechanism already exists in the current IOFSL implementation along with other mechanisms designed to handle/execute tasks efficiently.

On the other end, the client may issue requests and wait for the corresponding tasks to complete. As the requests are asynchronously transmitted, it is necessary to track their progress and wait for their completion. The main caller thread may monitor the completion of every I/O request that is issued; however if the number of requests becomes significant, the corresponding overhead to track/wait for the completion of these requests may increase (depending on whether or not they have completed already). Therefore the HDF5 client may also use an additional progress thread to track and wait for the completion of the requests in background, which may be an easy and quite beneficial optimization.

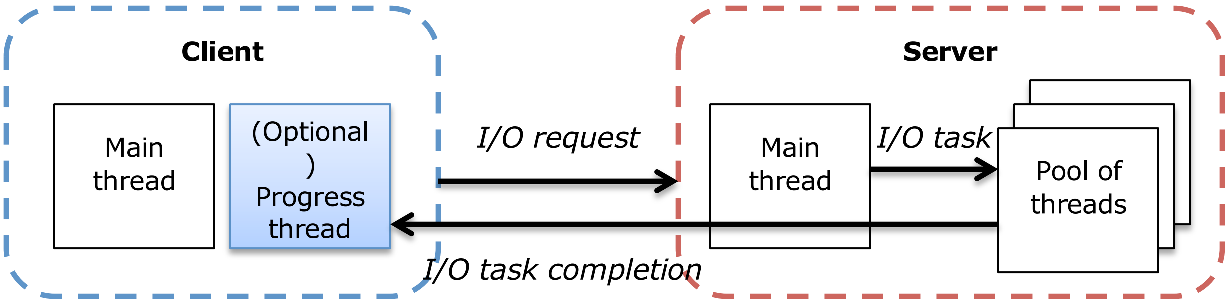


Figure 1. Multithreaded architecture.

2.2 Generic Function Forwarding

Shipping APIs from the compute nodes to the I/O nodes implies the definition of an interface, which can transmit function parameters and receive function results back. As function calls have variable number of arguments, every I/O call that is forwarded to the server will have its arguments and return results serialized and de-serialized before being mapped to the actual function call. An encoding/decoding mechanism such as XDR (Sun Microsystems, Inc., 1987) will be used for portability, when needed. Using this mechanism, the server may receive requests from the client with a given task ID, de-serialize the request, and use additional threads to execute the corresponding tasks.

A process of auto-generation will be used (e.g., macros) to remove the need for the user to have to manually write the serialization/de-serialization functions (which would cause significant code modifications). Generated functions will be stored and retrieved using a corresponding task ID. Using this generation process, any function along with its ID will be passed to the server, and the server to retrieve and decode the parameters using the generated functions associated to that ID.

2.3 Network Abstraction Layer

Before one can start shipping I/O calls, an efficient transport protocol must be chosen to take advantage of the underlying interconnect available on the system and communicate with the IOFSL server. This transport protocol may be different depending on the system and network type that links the compute nodes to the I/O nodes. The framework must therefore be built on top of an abstraction layer referred to as a *network abstraction layer*, which will allow multiple communication protocols to be added and selected without requiring modification of the IOFSL server and HDF5 client.

In a standard use case, the client may dynamically connect to the server and start shipping I/O calls. However this may not be feasible depending on the underlying network implementation used (see section 4). This layer must therefore also be defined to allow a dynamic or static type of communication between the server and the client so they can be launched as separate jobs or within the same job if the communication protocol (e.g., MPI on most large HPC systems) does not allow it.

Once a connection model defined, all the operations issued through this layer are asynchronous and mainly divided into two categories that reflect the nature of I/O calls: metadata operations and bulk data operations.

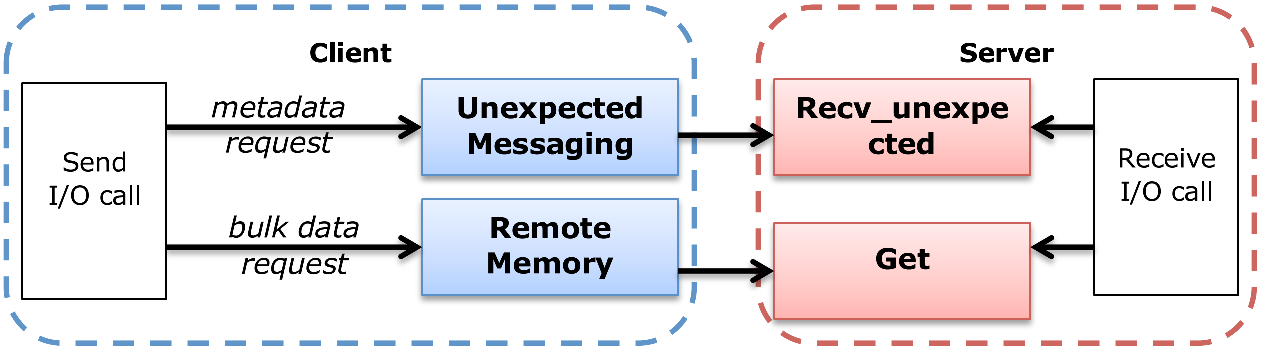


Figure 2. The network abstraction layer separates metadata from bulk data transfers.

Metadata and Bulk Data Operations

We make use of a mechanism that already exists in the IOFSL implementation to send and receive metadata operations to/from the IOFSL server: unexpected messaging. Every message that is sent using this mechanism does not require a matching receive operation to complete (i.e., it uses an eager protocol as opposed to a rendezvous protocol). This allows the IOFSL server to poll for messages in a message queue and process them without prior notification. To conserve resources on the server, we limit the use of this protocol for small size messages. Note that this size limit may vary depending on the network transport used but should be of the order of the kilobyte.

For larger messages (i.e., bulk data operations) the transfers will make use of remote memory access (RMA) protocols so that data can be transferred to the IOFSL server in a more scalable and efficient way. To control the data flow (and protect the server) a function shipping request will be sent to the server and the server will initiate the RMA operation (put or get operation) to pull or push data to/from the client. Moreover, as large scale systems now natively support RDMA, making use of this technique will decrease the CPU usage on both the client and the server.

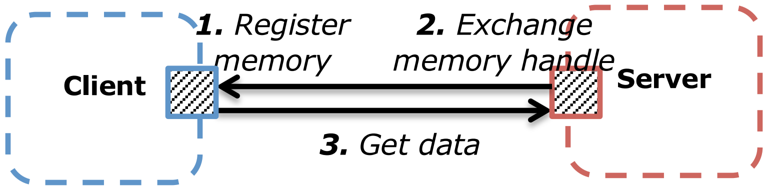


Figure 3. Bulk data transfers using RMA.

Memory must be registered to be remotely accessible; hence the corresponding handle (and/or memory location) must be exchanged before any operation can be executed on that memory region. Once this step done, put and get RMA operations can be issued. This mechanism will allow us to maximize the transfers between the client and the server without further involvement of the remote end (note that depending on the underlying implementation and network plugin developed, this may not be completely true). This capability is one of the crucial points of the framework.

2.4 Architecture Conclusion

By making use of the existing IOFSL architecture and adding new capabilities dedicated to the transfer of large data, the framework will be able to operate in an asynchronous and scalable manner while being able to interactively switch between metadata and bulk data transfers. The IOFSL server already having a proven scalability and several features such as multithreading of I/O requests and tasks, adding new capabilities such as a generic network transport layer and generic I/O call forwarding will allow us to re-use these features while incorporating new developments directly into the IOFSL project. This choice may therefore be particularly beneficial for the project.

3 Network Abstraction Layer Implementation

The network abstraction layer is one of the key parts of the framework since it is responsible of all the transfers between the client and the server. We detail here the API and some of the implementation choices.

A plug-in architecture that allows different network types to be implemented is chosen to make this layer as modular and flexible as possible. Two different plugins are currently being developed: one makes use of the original BMI implementation; and the other makes use of MPI. One of the main reasons for adding an MPI plugin is obvious: it allows us to take advantage of the underlying network implementation without any system specific development. More plugins will be added in the future depending on the project needs and network protocol limitations we may face (see section 4).

The network abstraction layer is organized as follows:

```
typedef size_t    na_size_t;
typedef void *    na_addr_t;
typedef int       na_tag_t;
typedef void *    na_request_t;
typedef void *    na_mem_handle_t;
typedef ptrdiff_t na_offset_t;
typedef struct    na_status_t {
    na_size_t count;
    /* more fields to be added */
} na_status_t;

typedef struct network_class_t {
    void (*finalize)(void);
    na_size_t (*get_unexpected_size)(void);

    /* Peer lookup */
    int (*lookup)(const char *name, na_addr_t *target);
    int (*free)(na_addr_t target);

    /* Metadata */
    int (*send_unexpected)(const void *buf, na_size_t buf_len,
            na_addr_t dest, na_tag_t tag,
            na_request_t *request, void *op_arg);
    int (*recv_unexpected)(void *buf, na_size_t *buf_len,
            na_addr_t *source, na_tag_t *tag,
            na_request_t *request, void *op_arg);

    int (*send)(const void *buf, na_size_t buf_len,
            na_addr_t dest, na_tag_t tag,
            na_request_t *request, void *op_arg);
    int (*recv)(void *buf, na_size_t buf_len,
            na_addr_t source, na_tag_t tag,
            na_request_t *request, void *op_arg);

    /* Bulk data */
    int (*mem_register)(void *buf, na_size_t buf_len, unsigned long flags,
na_mem_handle_t *mem_handle); /* flag to specify origin and operation allowed */
    int (*mem_deregister)(na_mem_handle_t mem_handle);

    int (*mem_handle_serialize)(void *buf, na_size_t buf_len, na_mem_handle_t
mem_handle);
    int (*mem_handle_deserialize)(na_mem_handle_t *mem_handle, const void *buf,
na_size_t buf_len);

    int (*put)(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);
    int (*get)(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);

    /* Progress */
    int (*wait)(na_request_t request, int *flag, int timeout, na_status_t
*status);
   /* More progress functions to be eventually added */
} network_class_t;
```

Peer lookup calls allow the client to look for a remote server using a name or string that identifies the server on the network. Metadata calls define both unexpected and expected messaging operations; bulk data calls define memory registration and put/get RMA operations. Note that in this case we also add handle serialization calls, which will be used to exchange memory handles over the network. Finally a wait call waits/tests for the completion of the asynchronous calls.

4 Unknowns and Limitations

While this framework defines a scalable and asynchronous interface for shipping I/O functions to a remote I/O node, some implementation choices are unknown or may present limitations.

4.1 Network Support

While the network abstraction layer can provide us with an easy and flexible way of switching network transport mechanisms, adding a degree of abstraction may slightly decrease the resulting performance of the transfers (this should be really minor as we operate at a very low level of abstraction). Moreover the BMI plugin, which only supports portals 3, may be replaced by another communication layer or by the MPI plugin, though this may in turn remove part of the functionality.

Part of this functionality is the dynamic connection of applications. BMI supports dynamic connection but MPI does not support it on all the large HPC systems. This issue may be solved in the future, although this remains a major unknown. The only way of using the MPI plugin on these systems will therefore imply setting aside nodes, which may in turn prevents multiple clients from connecting to the same IOFSL server and may create a potential resource constraint on the I/O node (as multiple IOFSL servers would have to be launched). Moreover this behavior is not desired for fault tolerance considerations, as a fault in the client will consequently interrupt the server.

Making use of RMA for the MPI plugin presents also some limitations. MPI 2 supports RMA operations but requires the creation of windows to be collective, which decreases the flexibility and usage one can make. It is worth noting that MPI 3 may fix this issue by introducing dynamic windows, although since it has only been released recently, the features/performance of the revised RMA API, which is to be implemented on these systems, will require further testing.

4.2 IOFSL

The IOFSL framework itself introduces another limitation that may appear in the future. Enhancing it and making changes to support generic I/O forwarding and other features such as bulk data transfers may lead to unknown potential issues and limitations.

4.3 I/O APIs

Another unknown resides in the I/O APIs that are to be shipped. While these APIs are still being defined, it may appear that it is more convenient and more efficient to directly ship HDF5-based calls instead of raw I/O API calls. This will have to be defined in the future, although the framework is being developed to allow generic forwarding of I/O calls. Additionally, when HDF5/IOD/DAOS are all running on 1 node, the function shipper must be elided completely and the I/O calls must complete by directly calling the appropriate routines, without passing through the function shipper.

5 Framework Demonstration

As a simple demonstration we propose to implement a client and a server applications that make use of the minimal and required set of functions necessary to ship calls over the network. Parameters are encoded and sent to the server asynchronously. The client makes use of a request ID to wait for the completion of these requests on the server and for the return parameters to be sent back.

Simplified client snippet:

```
int main(int argc, char *argv[])
{
    client_register();
    client_forward(in1, &out1, &req1);
    client_forward(in2, &out2, &req2);
    client_forward(in3, &out3, &req3);
    client_forward(in4, &out4, &req4);

    client_wait(req3, time_out);
    client_wait(req4, time_out);
    client_wait(req1, time_out);
    client_wait(req2, time_out);

// Could also have used:
//  client_wait_all({req1, req2, req3, rea4}, time_out);

    printf("Received op status: %d, %d, %d, %d\n", out1, out2, out3, out4);

    client_finalize();
}
```

Simplified server snippet:

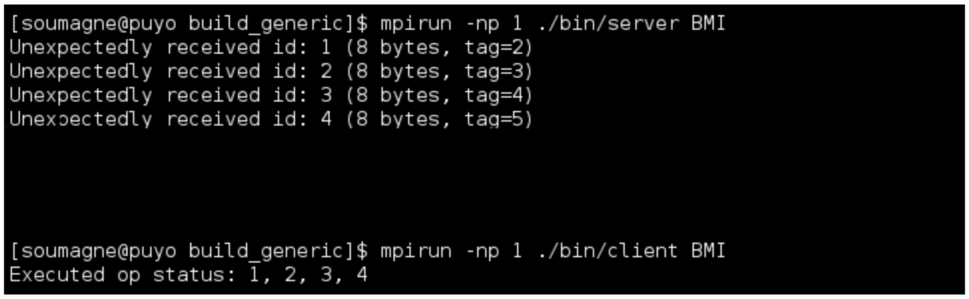
```
int main(int argc, char *argv[])
{
    if (strcmp("MPI", argv[1]) == 0) {
        na_mpi_init(NULL, MPI_INIT_SERVER);
    } else {
        char *listen_addr = getenv(ION_ENV);
        if (!listen_addr) {
            fprintf(stderr, "getenv(\"%s\") failed.\n", ION_ENV);
            return EXIT_FAILURE;
        }
        na_bmi_init("bmi_tcp", listen_addr, BMI_INIT_SERVER);
    }

    for (i = 0; i < 4; i++) {
        na_recv_unexpected(recv_buf, &recv_buf_len, &source, &tag, NULL);
        xdr_decode(recv_buf, recv_buf_len, &recv_id);
        printf("Unexpectedly received id: %d (%lu bytes, tag=%d)\n",
                (int) recv_id, (long unsigned int) recv_buf_len, tag);
        op_status = recv_id;
        xdr_encode(recv_buf, recv_buf_len, &op_status);
        na_send(recv_buf, recv_buf_len, source, tag, &req, NULL);
        na_wait(req, NA_BMI_MAX_IDLE_TIME, NA_STATUS_IGNORE);
    }

    na_finalize();
}
```

Test result:

Using the same test programs, two different tests are run, one using BMI and the other using MPI.

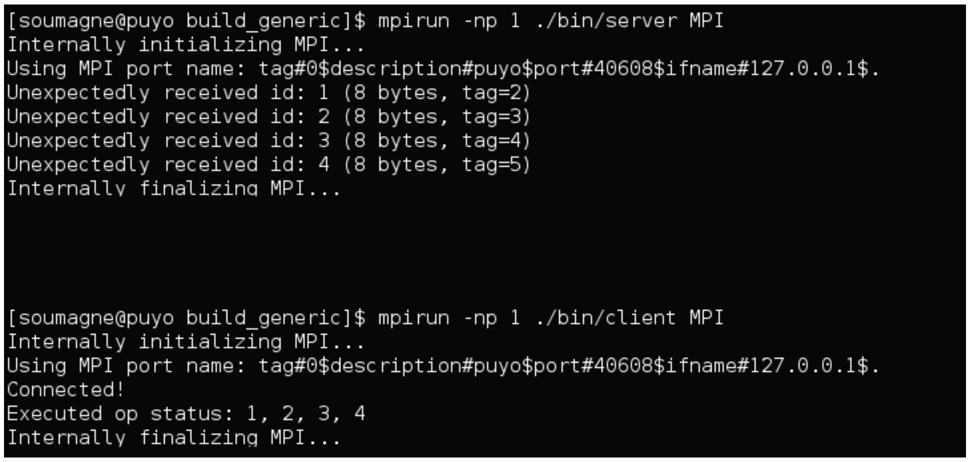


```
[soumagne@puyo build_generic]$ mpirun -np 1 ./bin/server BMI
Unexpectedly received id: 1 (8 bytes, tag=2)
Unexpectedly received id: 2 (8 bytes, tag=3)
Unexpectedly received id: 3 (8 bytes, tag=4)
Unexpectedly received id: 4 (8 bytes, tag=5)




[soumagne@puyo build_generic]$ mpirun -np 1 ./bin/client BMI
Executed op status: 1, 2, 3, 4
```

Figure 4. Client/server test using BMI.



```
[soumagne@puyo build_generic]$ mpirun -np 1 ./bin/server MPI
Internally initializing MPI...
Using MPI port name: tag#0$description#puyo$port#40608$ifname#127.0.0.1$.
Unexpectedly received id: 1 (8 bytes, tag=2)
Unexpectedly received id: 2 (8 bytes, tag=3)
Unexpectedly received id: 3 (8 bytes, tag=4)
Unexpectedly received id: 4 (8 bytes, tag=5)
Internally finalizing MPI...




[soumagne@puyo build_generic]$ mpirun -np 1 ./bin/client MPI
Internally initializing MPI...
Using MPI port name: tag#0$description#puyo$port#40608$ifname#127.0.0.1$.
Connected!
Executed op status: 1, 2, 3, 4
Internally finalizing MPI...
```

Figure 5. Client/server test using MPI.

6 Conclusion

The framework that we define allows calls to be forwarded to an IOFSL server using two distinct mechanisms: one dedicated to the shipping of metadata operations using unexpected messages and the other dedicated to the shipping of bulk data operations using remote memory access. The genericity of the approach allows HDF5 to ship any I/O operation directly to the I/O node where an IOFSL server operates in a multithreaded environment, allowing these operations to be asynchronously executed. By making use of the already proven scalability of IOFSL and enhancing it with new features, this framework will allow HDF5 to operate in an exascale environment, adding a small or null overhead on the compute node.

References

Ali, N., Carns, P. H., Kimpe, D., Lang, S., Latham, R., Ross, R. B., et al. (2009). Scalable I/O Forwarding Framework for High-Performance Computing Systems. *CLUSTER '09* (pp. 1-10). IEEE.

Birrell, A. D., & Nelson, B. J. (1984). Implementing Remote Procedure Calls. *ACM Trans. Comput. Syst. , 2* (1), 39-59.

Carns, P., Ligon, W. I., Ross, R., & Wyckoff, P. (2005). BMI: A Network Abstraction Layer for Parallel I/O. *19th IEEE International Parallel and Distributed Processing Symposium.*

Hosting, G. P. (2012). From Protocol Buffers - Google's data interchange format: http://code.google.com/p/protobuf

OMG. (2012). From CORBA: http://www.corba.org

Sun Microsystems, Inc. (1987, June). From RFC 1014 - XDR: External Data Representation Standard: http://tools.ietf.org/html/rfc1014