Mercury: Enabling Remote Procedure Call for High-Performance Computing

Jerome Soumagne*, Dries Kimpe†, Judicael Zounmevo‡, Mohamad Chaarawi*,
Quincey Koziol*, Ahmad Afsahi‡, Robert Ross†
*The HDF Group
Champaign, IL 61820
†Argonne National Laboratory
Argonne, IL 60439
‡ECE dept. Queen's University
19 Union Street, Kingston, ON, K7L 3N6, Canada

***Abstract*—Remote Procedure Call (RPC) is a technique that has been largely used by distributed services. This technique, now more and more used in the context of High-Performance Computing (HPC), allows the execution of routines to be delegated to remote nodes, which can be set aside and dedicated to specific tasks. However, existing RPC frameworks assume a sockets based network interface (usually on top of TCP/IP) which is not appropriate for HPC systems, as this API does not typically map well to the native network transport used on those systems, resulting in lower network performance. In addition, existing RPC frameworks often do not support handling large data arguments, such as those found in read or write calls.**

We present in this paper an asynchronous RPC interface specifically designed for use in HPC systems that allows asynchronous transfer of parameters and execution requests and direct support of large data arguments. The interface is generic to allow any function call to be shipped. Additionally, the network implementation is abstracted, allowing easy porting to future systems and efficient use of existing native transport mechanisms.

I. Introduction

When working in an heterogeneous environment, it is often very useful for an engineer or a scientist to be able to distribute the various steps of an application workflow; particularly so in high-performance computing where it is common to see systems or nodes embedding different types of resources and libraries, which can be dedicated to specific tasks such as computation, storage or analysis and visualization. Remote procedure call (RPC) [1] is a technique that follows a client/server model and allows local calls to be transparently executed onto remote resources. It consists of serializing the local function parameters into a memory buffer and sending that buffer to a remote target which in turn deserializes the parameters and executes the corresponding function call. Libraries implementing this technique can be found in various domains such as web services with Google Protocol Buffers [2] or Facebook Thrift [3], or in domains such as grid computing with GridRPC [4]. RPC can also be realized using a more object oriented approach with frameworks such as CORBA [5] or Java RMI [6] where abstract objects and methods can be distributed across a range of nodes or machines.

However, using these standard and generic RPC frameworks on an HPC system presents two main limitations: the inability to take advantage of the native transport mechanism to transfer data efficiently, as these frameworks are mainly designed on top of TCP/IP protocols; and the inability to transfer very large amounts of data, as the limit imposed by the RPC interface is generally of the order of the megabyte. In addition, even if no limit is enforced, transferring large amounts of data through the RPC library is usually discouraged, mostly due to overhead caused by serialization and encoding, causing the data to be copied many times before reaching the remote node.

The paper is organized as follows: we first discuss related work in section II, then in section III we discuss the network abstraction layer on top of which the interface is built, as well as the architecture defined to transfer small and large data efficiently. Section IV outlines the API and shows its advantages to enable the use of pipelining techniques. We then describe the development of network transport plugins for our interface as well as performance evaluation results. Section V presents conclusions and future work directions.

II. Related Work

The Network File System (NFS) [7] is a very good example of the use of RPC with large data transfers and therefore very close to the use of RPC on an HPC system. It makes use of XDR [8] to serialize arbitrary data structures and create a system-independent description, the resulting stream of bytes is then sent to a remote resource, which can deserialize and get the data back from it. It can also make use of separate transport mechanisms (on recent versions of NFS) to transfer data over RDMA protocols, in which case the data is processed outside of the XDR stream. The interface that we present in this paper follows similar principles but in addition handles bulk data directly. It also does not limit to the use of XDR for data encoding, which can be a performance hit, especially when sender and receiver share a common system architecture. By providing a *network abstraction layer*, the RPC interface that we define gives the ability to the user to send small data and large data efficiently, using either small messages or remote memory access (RMA) types of transfer that fully support one-sided semantics present on recent HPC systems. Furthermore, all the interface presented is non-blocking and therefore allows an asynchronous mode of operation, preventing the caller to wait for an operation to execute before another one can be issued.

The *I/O Forwarding Scalability Layer* (IOFSL) [9] is another project upon which part of the work presented in this paper is based. IOFSL makes use of RPC to specifically forward I/O calls. It defines an API called ZOIDFS that locally serializes function parameters and sends them to a remote server, where they can in turn get mapped onto file system specific I/O operations. One of the main motivations for extending the work that already exists in IOFSL is the ability to send not only a specific set of calls, as the ones that are defined through the ZOIDFS API, but a various set of calls, which can be dynamically and generically defined. It is also worth noting that IOFSL is built on top of the BMI [10] network transport layer used in the Parallel Virtual File System (PVFS) [11]. It allows support for dynamic connection as well as fault tolerance and also defines two types of messaging, unexpected and expected (described in section III-B), that can enable an asynchronous mode of operation. Nevertheless, BMI is limited in its design by not directly exposing the RMA semantics that are required to explicitly achieve RDMA operations from the client memory to the server memory, which can be an issue and a performance limitation (main advantages of using an RMA approach are described in section III-B). In addition, while BMI does not offer one-sided operations, it does provide a relatively high level set of network operations. This makes porting BMI to new network transports (such as the Cray Gemini interconnect [12]) to be a non-trivial work, and more time consuming than it should be, as only a subset of the functionality provided by BMI is required for implementing RPC in our context.

Another project, Sandia National Laboratories' *NEtwork Scalable Service Interface* (Nessie) [13] system provides a simple RPC mechanism originally developed for the Lightweight File Systems [14] project. It provides an asynchronous RPC solution, which is mainly designed to overlap computation and I/O. The RPC interface of Nessie directly relies on the Sun XDR solution which is mainly designed to communicate between heterogeneous architectures, even though practically all High-Performance Computing systems are homogeneous. Nessie provides a separate mechanism to handle bulk data transfers, which can use RDMA to transfer data efficiently from one memory to the other, and supports several network transports. The Nessie client uses the RPC interface to push control messages to the servers. Additionally, Nessie exposes a different, one-sided API (similar to Portals [15]), which the user can use to push or pull data between client and server. Mercury is different, in that it's interface, which also supports RDMA natively, can transparently handle bulk data for the user by automatically generating abstract memory handles representing the remote large data arguments, which are easier to manipulate and do not require any extra effort by the user. Mercury also provides fine grain control on the data transfer if required (for example to implement pipelining). In addition, Mercury provides a higher level interface than Nessie, greatly reducing the amount of user code needed to implement RPC functionality.

Another similar approach can be seen with the *Decoupled and Asynchronous Remote Transfers* (DART) [16] project. While DART is not defined as an explicit RPC framework, it allows transfer of large amounts of data using a client/server model from applications running on the compute nodes of a HPC system to local storage or remote locations, to enable remote application monitoring, data analysis, code coupling, and data archiving. The key requirements that DART is trying to satisfy include minimizing data transfer overheads on the application, achieving high-throughput, low-latency data transfers, and preventing data losses. Towards achieving these goals, DART is designed so that dedicated nodes, i.e., separate from the application compute nodes, asynchronously extract data from the memory of the compute nodes using RDMA. In this way, expensive data I/O and streaming operations from the application compute nodes to dedicated nodes are offloaded, and allow the application to progress while data is transferred. While using DART is not transparent and therefore requires explicit requests to be sent by the user, there is no inherent limitation for integration of such a framework within our network abstraction layer and therefore wrap it within the RPC layer that we define, hence allowing users to transfer data using DART on the platforms it supports.

III. Architecture

As mentioned in the previous section, Mercury's interface relies on three main components: a network abstraction layer, an RPC interface that is able to handle calls in a generic fashion and a bulk data interface, which complements the RPC layer and is intended to easily transfer large amounts of data by abstracting memory segments. We present in this section the overall architecture and each of its components.

A. Overview

The RPC interface follows a client / server architecture. As described in figure 1, issuing a remote call results in different steps depending on the size of the data associated with the call. We distinguish two types of transfers: transfers containing typical function parameters, which are generally small, referred to as *metadata*, and transfers of function parameters describing large amounts of data, referred to as *bulk data*.

Every RPC call sent through the interface results in the serialization of function parameters into a memory buffer (its size generally being limited to one kilobyte, depending on the interconnect), which is then sent to the server using the network abstraction layer interface. One of the key requirements is to limit memory copies at any stage of the transfer, especially when transferring large amounts data. Therefore, if the data sent is small, it is serialized and sent using small messages, otherwise a description of the memory region that is to be transferred is sent within this same small message to the server, which can then start pulling the data (if the data is the input of the remote call) or pushing the data (if the data is the output of the remote call). Limiting the size of the initial RPC request

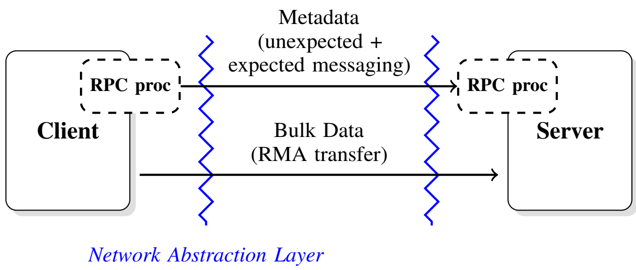


Fig. 1: Architecture overview: each side uses an RPC *processor* to serialize and deserialize parameters sent through the interface. Calling functions with relatively small arguments results in using the short messaging mechanism exposed by the network abstraction layer, whereas functions containing large data arguments additionaly use the RMA mechanism.

to the server also helps in scalability, as it avoids unnecessary server resource consumption in case of large numbers of clients concurrently accessing the same server. Depending on the degree of control desired, all these steps can be transparently handled by Mercury or directly exposed to the user.

B. Network Abstraction Layer

The main purpose of the *network abstraction layer* is as its name suggests to abstract the network protocols that are exposed to the user, allowing multiple transports to be integrated through a system of plugins. A direct consequence imposed by this architecture is to provide a lightweight interface, for which only a reasonable effort will be required to implement a new plugin. The interface itself must defines three main types of mechanisms for transferring data: unexpected messaging, expected messaging and remote memory access; but also the additional setup required to dynamically establish a connection between the client and the server (although a dynamic connection may not be always feasible depending on the underlying network implementation used).

Unexpected and expected messaging is limited to the transfer of short messages and makes use of a two-sided approach. The maximum message size is, for performance reasons, determined by the interconnect and can be as small as a few kilobytes. The concept of unexpected messaging is used in other communication protocols such as BMI [10]. Sending an unexpected message through the network abstraction layer does not require a matching receive to be posted before it can complete. By using this mechanism, clients are not blocked and the server can, every time an unexpected receive is issued, pick up the new messages that have been posted. Another difference between expected and unexpected messages is unexpected messages can arrive from any remote source, while expected messages require the remote source to be known.

The remote memory access (RMA) interface allows remote memory chunks (contiguous and non-contiguous) to be accessed. In most one-sided interfaces and RDMA protocols, memory must be registered to the network interface controller (NIC) before it can used. The purpose of the interface defined in the network abstraction layer is to create a first degree of abstraction and define an API that is compatible with most RMA protocols. Registering a memory segment to the NIC typically results in the creation of a handle to that segment containing virtual address information, etc. The local handle created needs to be communicated to the remote node before that node can start a put or get operation. The network abstraction is responsible for ensuring that these memory handles can be serialized and transferred across the network. Once handles are exchanged, a non-blocking put or get can be initiated. On most interconnects, put and get will map to the put and get operation provided by the specific API provided by the interconnect. The network abstraction interface is designed to allow the emulation of one-sided transfers on top of two-sided sends and receives for network protocols such as TCP/IP that only support a two-sided messaging approach.

With this network abstraction layer in place, Mercury can easily be ported to support new interconnects. The relatively limited functionality provided by the network abstraction (for example, no unlimited size two-sided messages) ensures close to native performance.

C. RPC Interface and Metadata

Sending a call that only involves small data makes use of the unexpected / expected messaging defined in III-B. However, at a higher level, sending a function call to the server means concretely that the client must know how to encode the input parameters before it can start sending information and know how to decode the output parameters once it receives a response from the server. On the server side, the server must also have knowledge of what to execute when it receives an RPC request and how it can decode and encode the input and output parameters. The framework for describing the function calls and encoding/decoding parameters is key to the operation of our interface.

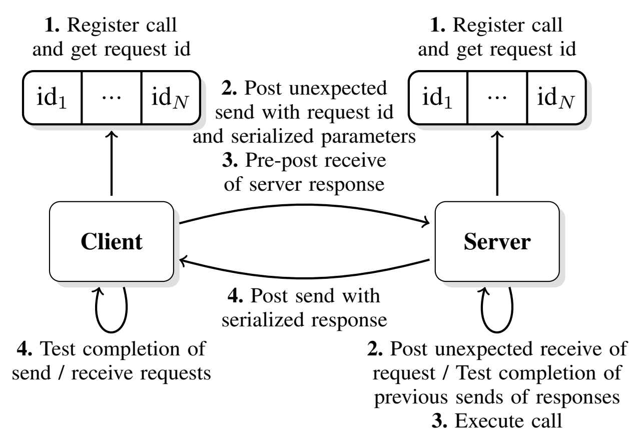


Fig. 2: Asynchronous execution flow of RPC call. The receive buffer is pre-posted, allowing the client to get other work done while the call is remotely executed and the response is sent back.

One of the important points is the ability to support a set of function calls that can be sent to the server in a generic fashion, avoiding the limitations of a hard-coded set of routines. The generic framework is described in figure 2. 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 client encodes the input parameters along with that operations ID into a buffer and send it to the server using an unexpected messaging protocol, which is non-blocking. To ensure full asynchrony, the memory buffer used to receive the response back from the server is also pre-posted by the client. For reasons of efficiency and resource consumption, these messages are limited in size (typically a few kilobytes). However if the metadata exceeds the size of an unexpected message, the client will need to transfer the metadata in separate messages, making transparent use of the bulk data interface described in III-D to expose the additional metadata to the server.

When the server receives a new request ID, it looks up the corresponding callback, decodes the input parameters, executes the function call, encodes the output parameters and starts sending the response back to the client. Sending a response back to the client is also non-blocking, therefore, while receiving new function calls, the server can also test a 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.

With this mechanism in place, it becomes simple to extend it to handle bulk data.

D. Bulk Data Interface

In addition to the previous interface, some function calls may require the transfer of larger amounts of data. For these function calls, the bulk data interface is used and is built on top of the remote memory access protocol defined in the network abstraction layer. Only the RPC server initiates one-sided transfers so that it can, as well as controlling the data flow, protect its memory from concurrent accesses.

As described in figure 3, the bulk data transfer interface uses a one-sided communication approach. The RPC client exposes a memory region to the RPC server by creating a bulk data descriptor (which contains virtual memory address information, size of the memory region that is being exposed, and other parameters that may depend on the underlying network implementation). The bulk data descriptor can then be serialized and sent to the RPC server along with the RPC request parameters (using the RPC interface defined in section III-C). When the server decodes the input parameters,

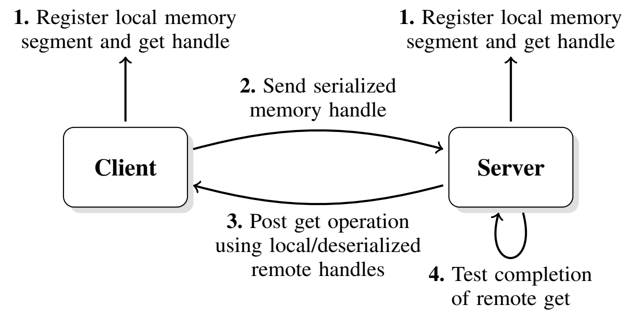


Fig. 3: Transfer mechanism when doing an RPC requires large data to be sent to the server.

it deserializes the bulk data descriptor and gets the size of the memory buffer that has to be transferred.

In the case of an RPC request that *consumes* large data parameters, the RPC server may allocate a buffer of the size of the data that needs to be received, expose its local memory region by creating a bulk data block descriptor and initiate an asynchronous read / get operation on that memory region. The RPC server then waits / tests for the completion of the operation and executes the call once the data has been fully received (or partially if the execution call supports it). The response (i.e., the result of the call) is then sent back to the RPC client and memory handles are freed.

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

Transferring data through this process can be transparent for the user, especially since the RPC interface can also take care of serializing / deserializing the memory handles along with the other parameters. This is particularly important when non-contiguous memory segments have to be transferred. In either case memory segments are automatically registered on the RPC client and are abstracted by the memory handle created. The memory handle is then serialized along with the parameters of the RPC function and transferring large data using non-contiguous memory regions therefore results in the same process described above. Note that the handle may be variable size in this case as it may contain more information and also depends on the underlying network implementation that can support registration of memory segments directly.

IV. EVALUATION

The architecture previously defined enables generic RPC calls to be shipped along with handles that can describe contiguous and non-contiguous memory regions when a bulk data transfer is required. We present in this section how one can take advantage of this architecture to build a pipelining mechanism that can easily request blocks of data on demand.

A. Pipelining Bulk Data Transfers

Pipelining transfers is a typical use case when one wants to overlap communication and execution. In the architecture that we described, requesting a large amount of data to be processed results in an RPC request being sent from the RPC client to the RPC server as well as a bulk data transfer. In a common use case, the server may wait for the entire data to be received before executing the requested call. However, by pipelining the transfers, one can in fact start processing the data while it is being transferred, avoiding to pay the cost of the latency for an entire RMA transfer. Note that although we focus on this point in the example below, using this technique can also be particularly useful if the RPC server does not have enough memory to handle all the data that needs to be sent, in which case it will also need to transfer data as it processes it.

A simplified version of the RPC client code is presented below:

```
#define BULK_NX 16
#define BULK_NY 128

int main(int argc, char *argv[])
{
  hg_id_t rpc_id;
  write_in_t in_struct;
  write_out_t out_struct;
  hg_request_t rpc_request;
  int buf[BULK_NX][BULK_NY];
  hg_bulk_segment_t segments[BULK_NX];
  hg_bulk_t bulk_handle = HG_BULK_NULL;

  /* Initialize the interface */
  [...]
  /* Register RPC call */
  rpc_id = HG_REGISTER("write",
    write_in_t, write_out_t);

  /* Provide data layout information */
  for (i = 0; i < BULK_NX ; i++) {
    segments[i].address = buf[i];
    segments[i].size = BULK_NY * sizeof(int);
  }

  /* Create bulk handle with segment info */
  HG_Bulk_handle_create_segments(segments,
    BULK_NX, HG_BULK_READ_ONLY, &bulk_handle);

  /* Attach bulk handle to input parameters */
  [...]
  in_struct.bulk_handle = bulk_handle;

  /* Send RPC request */
  HG_Forward(server_addr, rpc_id,
    &in_struct, &out_struct, &rpc_request);

  /* Wait for RPC completion and response */
  HG_Wait(rpc_request, HG_MAX_IDLE_TIME,
    HG_STATUS_IGNORE);

  /* Get output parameters */
  [...]
  ret = out_struct.ret;

  /* Free bulk handle */
  HG_Bulk_handle_free(bulk_handle);

  /* Finalize the interface */
  [...]
}
```

When the client initializes, it registers the RPC call it wants to send. Because this call involves non contiguous bulk data transfers, memory segments that describe the memory regions are created and registered. The resulting `bulk_handle` is then passed to the `HG_Forward` call along with the other call parameters. One may then wait for the response and free the bulk handle when the request has completed (a notification may also be sent in the future to allow the bulk handle to be freed earlier, and hence the memory to be unpinned).

The pipelining mechanism happens on the server, which takes care of the bulk transfers. The pipeline itself has here a fixed pipeline size and a pipeline buffer size. A simplified version of the RPC server code is presented below:

```
#define PIPELINE_BUFFER_SIZE 256
#define PIPELINE_SIZE 4

int rpc_write(hg_handle_t handle)
{
  write_in_t in_struct;
  write_out_t out_struct;
  hg_bulk_t bulk_handle;
  hg_bulk_block_t bulk_block_handle;
  hg_bulk_request_t bulk_request[PIPELINE_SIZE];
  void *buf;
  size_t nbytes, nbytes_read = 0;
  size_t start_offset = 0;

  /* Get input parameters and bulk handle */
  HG_Handler_get_input(handle, &in_struct);
  [...]
  bulk_handle = in_struct.bulk_handle;

  /* Get size of data and allocate buffer */
  nbytes = HG_Bulk_handle_get_size(bulk_handle);
  buf = malloc(nbytes);

  /* Create block handle to read data */
  HG_Bulk_block_handle_create(buf, nbytes,
    HG_BULK_READWRITE, &bulk_block_handle);

  /* Initialize pipeline and start reads */
  for (p = 0; p < PIPELINE_SIZE; p++) {
    size_t offset = p * PIPELINE_BUFFER_SIZE;
    /* Start read of data chunk */
    HG_Bulk_read(client_addr, bulk_handle,
      offset, bulk_block_handle, offset,
      PIPELINE_BUFFER_SIZE, &bulk_request[p]);
  }

  while (nbytes_read != nbytes) {
    for (p = 0; p < PIPELINE_SIZE; p++) {
      size_t offset = start_offset +
        p * PIPELINE_BUFFER_SIZE;
      /* Wait for data chunk */
      HG_Bulk_wait(bulk_request[p],
        HG_MAX_IDLE_TIME, HG_STATUS_IGNORE);
      nbytes_read += PIPELINE_BUFFER_SIZE;

      /* Do work (write data chunk) */
      write(buf + offset, PIPELINE_BUFFER_SIZE);

      /* Start another read */
      offset += PIPELINE_BUFFER_SIZE *
        PIPELINE_SIZE;
      if (offset < nbytes) {
```

```
        HG_Bulk_read(client_addr,
          bulk_handle, offset,
          bulk_block_handle, offset,
          PIPELINE_BUFFER_SIZE,
          &bulk_request[p]);
      } else {
        /* Start read with remaining piece */
      }
    }
    start_offset += PIPELINE_BUFFER_SIZE
      * PIPELINE_SIZE;
  }

  /* Free block handle */
  HG_Bulk_block_handle_free(bulk_block_handle);
  free(buf);

  /* Start sending response back */
  [...]
  out_struct.ret = ret;
  HG_Handler_start_output(handle, &out_struct);
}

int main(int argc, char *argv[])
{
  /* Initialize the interface */
  [...]
  /* Register RPC call */
  HG_HANDLER_REGISTER("write", rpc_write,
    write_in_t, write_out_t);

  while (!finalized) {
    /* Process RPC requests (non-blocking) */
    HG_Handler_process(0, HG_STATUS_IGNORE);
  }

  /* Finalize the interface */
  [...]
}
```

Every RPC server, once it is initialized, must loop over a `HG_Handler_process` call, which waits for new RPC requests and executes the corresponding registered callback (in the same thread or new thread depending on user needs). Once the request is deserialized, the `bulk_handle` parameter can be used to get the total size of the data that is to be transferred, allocate a buffer of the appropriate size and start the bulk data transfers. In this example, the pipeline size is set to 4 and the pipeline buffer size is set to 256, which means that 4 RMA requests of 256 bytes are initiated. One can then wait for the first piece of 256 bytes to arrive and process it. While it is being processed, other pieces may arrive. Once one piece is processed a new RMA transfer is started for the piece that is at stage 4 in the pipeline and one can wait for the next piece, process it. Note that while the memory region registered on the client is non-contiguous, the `HG_Bulk_read` call on the server presents it as a contiguous region, simplifying server code. In addition, logical offsets (relative to the beginning of the data) can be given to move data pieces individually, with the bulk data interface taking care of mapping from the continuous logical offsets to the non-contiguous client memory regions.

We continue this process until all the data has been read / processed and the response (i.e., the result of the function call) can be sent back. Again we only start sending the response by calling the `HG_Handler_start_output` call and its completion will only be tested by calling `HG_Handler_process`, in which case the resources associated to the response will be freed. Note that all functions support asynchronous execution, allowing Mercury to be used in event driven code if so desired.

B. Network Plugins and Testing Environment

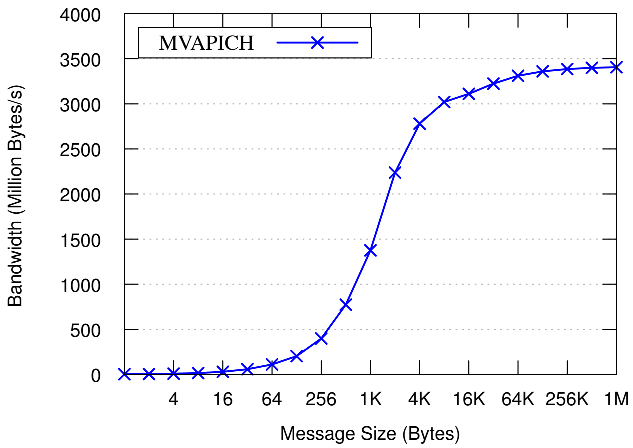
Two plugins have been developed as of the date this paper is written to illustrate the functionality of the network abstraction layer. At this point, the plugins have not been optimized for performance. One is built on top of BMI [10]. However, as we already pointed out in section II, BMI does not provide RMA semantics to efficiently take advantage of the network abstraction layer defined and the one-sided bulk data transfer architecture. The other one is built on top of MPI [17], which has only been providing full RMA semantics [18] recently with MPI3 [19]. Many MPI implementations, specifically those delivered with already installed machines, do not yet provide all MPI3 functionality. As BMI has not yet been ported to recent HPC systems, to illustrate the functionality and measure early performance results, we only consider the MPI plugin in this paper. This plugin, to be able to run on existing HPC systems limited to MPI-2 functionality, such as Cray systems, implements bulk data transfers on top of two-sided messaging. In practice, this means that for each bulk data transfer, an additional bulk data control message needs to be sent to the client to request either sending or receiving data. Progress on the transfer can then be realized by using a progress thread or by entering progress functions.

For testing we make use of two different HPC systems. One is an Infiniband QDR 4X cluster with MVAPICH [20] 1.8.1, the other one is a Cray XE6 with Cray MPT [21] 5.6.0.

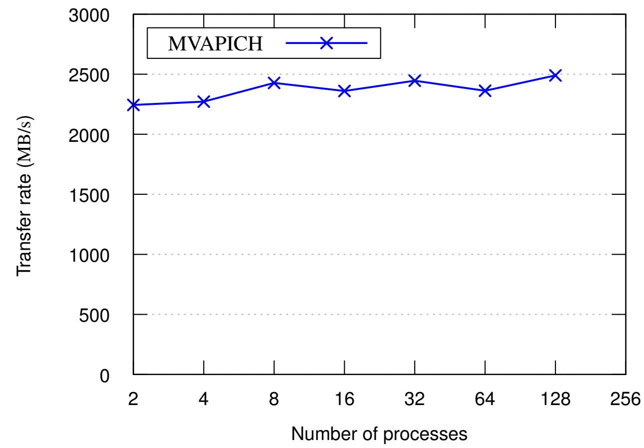
C. Performance Evaluation

As a first experiment, we measured the time it takes to execute a small RPC call (without any bulk data transfer involved) for an empty function (i.e., a function that returns immediately). On the Cray XE6 machine, measuring the average time for 20 RPC invocations, each call took $23\,\mu s$. This time includes the XDR encoding and decoding of the parameters of the function. However, as pointed out earlier, most HPC systems are homogeneous and thus don't require the data portability provided by XDR. When disabling XDR encoding (performing a simple memory copy instead) the time drops to $20\,\mu s$. This non-negligible improvement (15%) demonstrates the benefit of designing an RPC framework specifically for HPC environments.

The second experiment consists of testing the pipelining technique for bulk data transfers previously explained between one client and one server. As shown in table I, on Cray XE6 pipelining transfers can be particularly efficient when requests have already completed while other pipeline stages are being processed, allowing us to get very high bandwidth. However, the high injection bandwidth on this system makes it difficult to get good performance for small packets (such as the bulk data control messages due to the emulation of one-sided features on this system) particularly when data flow is not continuous.

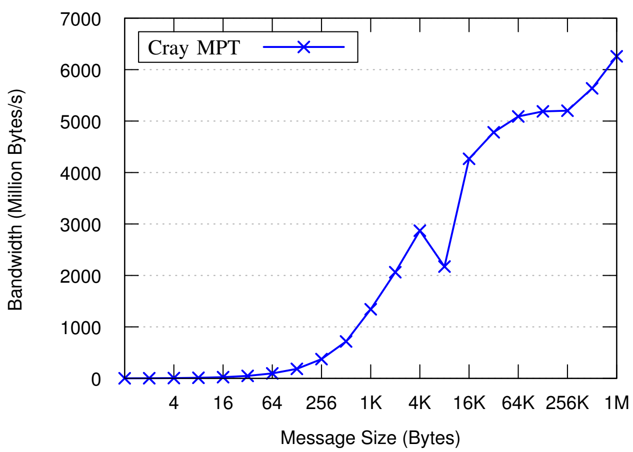


(a) Point-to-point internode bandwidth using OSU benchmark.

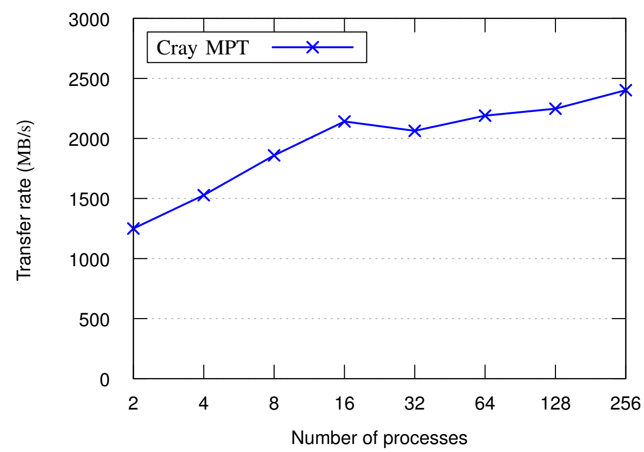


(b) Aggregate bandwidth of RPC request with 16 MB bulk data transfer and 4 MB pipeline buffer size.

Fig. 4: Bandwidth test on QDR 4X InfiniBand cluster using MVAPICH.



(a) Point-to-point internode bandwidth using OSU benchmark (Switch FMA/BTE for 8 kB message size).



(b) Aggregate bandwidth of RPC request with 512 MB bulk data transfer and 16 MB pipeline buffer size.

Fig. 5: Bandwidth test on Cray XE6 using Cray MPT.

TABLE I: Bandwidth of bulk data transfers using pipelining technique on Cray XE6 and a total buffer size of 16 MB.

Pipeline buffer size (kB)	Time (s)	Bandwidth (MB/s)
4096	0.009790	1634.25
2048	0.002927	5466.76
1024	0.014937	1071.16
512	0.002984	5362.62
256	0.003204	4993.14
128	0.013967	1145.57
64	0.005027	3182.69
32	0.018209	878.69
16	0.024311	658.14
8	0.051070	313.30
4	0.043660	366.46

Finally we evaluated the scalability of the RPC server by evaluating the total data throughput while increasing the number of clients. Figures 4 and 5 show the results for a QDR InfiniBand system (using MVAPICH) and the Cray XE6 system respectively. In both cases, in part due to the server side bulk data flow control mechanism, Mercury shows excellent scalability, with throughput either increasing or remaining stable as the number of concurrent clients increases. For comparison, the point to point message bandwidth on each system is shown. On the InfiniBand system, Mercury achieves about 70% of maximum network bandwidth. This is an excellent result, considering that the Mercury time represents an RPC call in addition to the data transfer, compared to the time to send a single message for the OSU benchmark. On the Cray system, performance is less good (about 40% of peak). We expect that this is mainly due to the relatively poor small message performance of the system, combined with the extra control messages caused by the one-sided emulation. However, it is also possible that the low performance is caused by a system limitation, considering that Nessie's performance for a similar operation (read) [22] shows the same low bandwidth, even though it is using true RDMA by bypassing MPI and using the interconnect's native uGNI API instead.

V. Conclusion and Future Work

In this paper we presented the Mercury framework. Mercury is specifically designed to offer RPC services in a High-Performance Computing environment. Mercury builds on a small, easily ported network abstraction layer providing operations closely matched to the capabilities of contemporary HPC network environments. Unlike most other RPC frameworks, Mercury offers direct support for handling remote calls containing large data arguments. Mercury's network protocol is designed to scale to thousands of clients. We demonstrated the power of the framework by implementing a remote *write* function including pipelining of large data arguments. We subsequently evaluated our implementation on two different HPC systems, showing both single client performance and multi-client scalability.

With the availability of the high-performing, portable, generic RPC functionality provided by Mercury, IOFSL can be simplifed and modernized by replacing the internal, hard coded IOFSL code by Mercury calls. As the network abstraction layer on top of which Mercury is built already supports using BMI for network connectivity, existing deployments of IOFSL continue to be supported, at the same time taking advantage of the improved scalability and performance of Mercury's network protocol.

Currently, Mercury does not offer support for canceling ongoing RPC calls. Cancellation is important for resiliency in environments where nodes or network can fail. Future work will include support for cancellation.

While Mercury already supports all required functionality to efficiently execute RPC calls, the amount of user code required for each call can be further reduced. Future versions of Mercury will provide a set of preprocessor macros, reducing the user's effort by automatically generating as much boiler plate code as possible.

The network abstraction layer currently has plugins for BMI, MPI-2 and MPI-3. However, as MPI RMA functionality is difficult to use in a client/server context [23], we intend to add native support for Infiniband networks, and the Cray XT and IBM BG/P and Q networks.

Acknowledgment

The work presented in this paper was supported by the Exascale FastForward project, LLNS subcontract no. B599860, and by the Office of Advanced Scientific Computer Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

References

[1] A. D. Birrell and B. J. Nelson, "Implementing Remote Procedure Calls," *ACM Trans. Comput. Syst.*, vol. 2, no. 1, pp. 39–59, Feb. 1984.

[2] Google Inc, "Protocol Buffers," 2012. [Online]. Available: https://developers.google.com/protocol-buffers

[3] M. Slee, A. Agarwal, and M. Kwiatkowski, "Thrift: Scalable Cross-Language Services Implementation," 2007.

[4] K. Seymour, H. Nakada, S. Matsuoka, J. Dongarra, C. Lee, and H. Casanova, "Overview of GridRPC: A Remote Procedure Call API for Grid Computing," in *Grid Computing—GRID 2002*, ser. Lecture Notes in Computer Science, M. Parashar, Ed. Springer Berlin Heidelberg, 2002, vol. 2536, pp. 274–278.

[5] Object Management Group, "Common Object Request Broker Architecture (CORBA)," 2012. [Online]. Available: http://www.omg.org/spec/CORBA

[6] A. Wollrath, R. Riggs, and J. Waldo, "A Distributed Object Model for the Java^{TM}System," in *Proceedings of the 2nd conference on USENIX Conference on Object-Oriented Technologies (COOTS) - Volume 2*, ser. COOTS'96. Berkeley, CA, USA: USENIX Association, 1996, pp. 17–17.

[7] R. Sandberg, D. Golgberg, S. Kleiman, D. Walsh, and B. Lyon, "Innovations in internet working," C. Partridge, Ed. Norwood, MA, USA: Artech House, Inc., 1988, ch. Design and Implementation of the Sun Network Filesystem, pp. 379–390.

[8] Sun Microsystems Inc, "RFC 1014—XDR: External Data Representation Standard," 1987. [Online]. Available: http://tools.ietf.org/html/rfc1014

[9] N. Ali, P. Carns, K. Iskra, D. Kimpe, S. Lang, R. Latham, R. Ross, L. Ward, and P. Sadayappan, "Scalable I/O forwarding framework for high-performance computing systems," in *IEEE International Conference on Cluster Computing and Workshops 2009*, ser. CLUSTER '09, 2009, pp. 1–10.

[10] P. Carns, I. Ligon, W., R. Ross, and P. Wyckoff, "BMI: a network abstraction layer for parallel I/O," in *19th IEEE International Parallel and Distributed Processing Symposium*, 2005.

[11] P. H. Carns, W. B. Ligon, III, R. B. Ross, and R. Thakur, "PVFS: A Parallel File System for Linux Clusters," in *In Proceedings of the 4th Annual Linux Showcase and Conference*. USENIX Association, 2000, pp. 317–327.

[12] R. Alverson, D. Roweth, and L. Kaplan, "The Gemini System Interconnect," in *IEEE 18th Annual Symposium on High-Performance Interconnects*, ser. HOTI, 2010, pp. 83–87.

[13] J. Lofstead, R. Oldfield, T. Kordenbrock, and C. Reiss, "Extending Scalability of Collective IO Through Nessie and Staging," in *Proceedings of the Sixth Workshop on Parallel Data Storage*, ser. PDSW '11. New York, NY, USA: ACM, 2011, pp. 7–12.

[14] R. Oldfield, P. Widener, A. Maccabe, L. Ward, and T. Kordenbrock, "Efficient Data-Movement for Lightweight I/O," in *Cluster Computing, 2006 IEEE International Conference on*, 2006, pp. 1–9.

[15] R. Brightwell, T. Hudson, K. Pedretti, R. Riesen, and K. Underwood, "Implementation and Performance of Portals 3.3 on the Cray XT3," in *Cluster Computing, 2005. IEEE International*, 2005, pp. 1–10.

[16] C. Docan, M. Parashar, and S. Klasky, "Enabling High-speed Asynchronous Data Extraction and Transfer Using DART," *Concurr. Comput. : Pract. Exper.*, vol. 22, no. 9, pp. 1181–1204, Jun. 2010.

[17] W. Gropp, E. Lusk, and R. Thakur, *Using MPI-2: Advanced Features of the Message-Passing Interface*. Cambridge, MA: MIT Press, 1999.

[18] W. Gropp and R. Thakur, "Revealing the Performance of MPI RMA Implementations," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, ser. Lecture Notes in Computer Science, F. Cappello, T. Herault, and J. Dongarra, Eds. Springer Berlin / Heidelberg, 2007, vol. 4757, pp. 272–280.

[19] "Message Passing Interface Forum," September 2012, MPI-3: Extensions to the message-passing interface. [Online]. Available: http://www.mpi-forum.org/docs/docs.html

[20] The Ohio State University, "MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE." [Online]. Available: http://mvapich.cse.ohio-state.edu/index.shtml

[21] H. Pritchard, I. Gorodetsky, and D. Buntinas, "A uGNI-Based MPICH2 Nemesis Network Module for the Cray XE," in *Recent Advances in the Message Passing Interface*, ser. Lecture Notes in Computer Science, Y. Cotronis, A. Danalis, D. Nikolopoulos, and J. Dongarra, Eds. Springer Berlin / Heidelberg, 2011, vol. 6960, pp. 110–119.

[22] R. A. Oldfield, T. Kordenbrock, and J. Lofstead, "Developing integrated data services for cray systems with a gemini interconnect," Sandia National Laboratories, Tech. Rep., 2012.

[23] J. A. Zounmevo, D. Kimpe, R. Ross, and A. Afsahi, "On the use of MPI in High-Performance Computing Services," p. 6, 2013.

(this page will be removed in the camera ready version)

Government License