RWAPI over InfiniBand: Design and Performance

Ouissem Ben Fredj and Éric Renault
GET / INT
Département Informatique
9, rue Charles Fourier
91011 Évry, France
Tel: +33 1 60 76 45 73 — Fax: +33 1 60 76 47 80
{ ouissem.benfredj, eric.renault } @int-evry.fr

Abstract

This paper presents the design of the minimalist communication interface called RWAPI over the InfiniBand interconnect for clusters of PCs. RWAPI has been developed to provide performance to higher applications on a wide variety of architectures. Since the specifications of the InfiniBand interconnect provides many ways to transfer data, we are discussing some issues regarding the choices between InfiniBand capabilities. We implemented RWAPI on top of the GRWA architecture and evaluated the communication performance. We obtained a very low latency and a throughput very close to the maximum user bandwidth for messages as small as 4 kilo-bytes.

Keywords: Programming model, cluster and grid computing, high-speed networks, performance.

1 Introduction

High-speed network interconnects that offer low latency and high bandwidth have been one of the main reasons attributed to the success of commodity cluster systems. Some of the leading high-speed networking interconnects include Gigabit-Ethernet, InfiniBand [3], Myrinet [7] and Quadrics. Two common features shared by these interconnects are User-level networking and Direct Memory Access (DMA). The best suited communication protocol that use efficiently these new features is the one-sided protocol. It means that the completion of a send (resp. receive) operation does not require the intervention of the receiver (resp. sender) process to take a complementary action. RDMA should be used to copy data to (from) the remote user space directly. Suppose that the receiver process have allocated a buffer to room incoming data and the sender have allocated a send buffer. Prior to the data transfer, the receiver must have sent its buffer address to the sender. Once the sender owns the destination address, it initiates a direct-deposit data sending. This task does not interfere with the receiver process. On the receiver side, it keeps on doing computation tasks, testing if new messages have arrived, or blocking until an incoming message event arises.

The need for a one-sided communication protocol has been recognized for many years. Some of these issues were initially addressed by the POrtable Run-Time Systems (PORTS) consortium [9]. One of the missions of PORTS was to define a standard API for one-sided communications. During the development, several different approaches were taken towards the one-sided API. The first one is the thread-to-thread communication paradigm which is supported by CHANT [15]. The second one is the remote service request (RSR) communication approach supported by libraries such NEXUS and DMCS. The third approach is a hybrid communication (that combines both prior paradigms) supported by the TULIP [6] project. These paradigms are widely used. For example, NEXUS supports the grid computing software infrastructure GLOBUS. MOL [8] extends DMCS with an object migration mechanism and a global namespace for the system. DMCS/MOL is used both in Parallel Runtime Environment for Multi-computer Applications (PREMA) [4] and in the Dynamic Load Balancing Library (DLBL).

In 1997, MPI-2 [16] (a new MPI standard) have been including some basic one-sided functionalities. Although, many studies have integrated one-sided communications to optimize MPI [11]. In 1999, a new communication library called Aggregate Remote Memory Copy Interface (ARMCI) [17] has been released. ARMCI is a high-level library designed for distributed array libraries and compiler run-time systems. IBM have maintained a low-level API, named LAPI [10], implementing the one-sided protocol and running on IBM SP systems only. Similarly, Cray SHMEM [5] provides direct send routines.
At the network layer, many factories have built RDMA features that ease the implementation of one-sided paradigms. For example, the HSL [24] network uses the PCI-Direct Deposit Component (PCI-DDC) [14] to offer a message-passing multiprocessor architecture based on a one-sided protocol. InfiniBand [3] proposes native one-sided communications. Myrinet [7, 2] and QNIX [23] do not provide native one-sided communications. But these features may be added (as for example in GM [1] with Myrinet since Myrinet NICs are programmable).

The arrival of these kind of networks has imposed common message-passing libraries to support RDMA (RW, GM, VIA [22]...). Most of these libraries have extended with one-sided communications to exploit RDMA features. But they do not use these functionalities as a base for their programming model.

2 RWAPI

Remote Write is a simple communication protocol in which the sender of a message is in charge of providing all the information needed to copy a contiguous memory area from one node to another.

![Figure 1. Message transfer with remote-write.](image)

Fig. 1 shows the different steps involved in a message transfer using the remote write:

1. the application writes the content of the message in a previously allocated contiguous memory area.

2. the application adds a new entry in the send queue. All information required to perform the message transfer are provided: local address (where the content of the message is located), remote address (where the message will be copied on the receiver) and the size of the message.

3. the application informs the NIC that a new entry have posted in the send queue.

4. the NIC read the next available entry in the send queue.

5. using the local address and the size of the message, the NIC reads the message content from memory...

6. and sends it to the destination node using the network.

7. when the message arrives on the destination node, it is copied in memory using the remote address and the size of the message.

8. once the message has been received completely, the NIC adds a new entry in the receive event queue.

9. the application is then able to read the content of the receive event queue to take into account new income messages.

10. when a message has been taken into account, the application notifies the NIC to free resources in the receive event queue.

RWAPI (which stands for Remote-Write Application Programming Interface) is a lightweight interface designed to provide a minimalist interface for the single remote-write primitive. The goal we are trying to achieve is to provide the smallest set of functions that enables to write any parallel programs. This way, we expect:

- to achieve the best performance for communications.
- to require as less development as possible to port our interface to new architectures (and GRWA — see next section — has been defined in this way).

There are two kinds of messages in RWAPI. The first message type requires the destination node identification, both local and remote addresses and the size of the message. Messages in this case can be of any length. The second message type just requires the destination node identification and the message content; these messages do not require any address or size, but are limited to a few 16 bytes. They may be helpfully used to transfer small amounts of information of any kind from one node to another. However, even if they are limited to this specific use, they are especially useful to exchange addresses before the other message type transfers can occur.

The API is as follows:

- int rwapi_init ( int, char ** ) must be called before any other RWAPI functions in order to set up the communication interface.
- int rwapi_finalize ( ) should be called after any other RWAPI functions and before exiting the program. This function ensures that all FIFOs are flushed before leaving.
• int rwapi_rank ( ) returns the rank of the local node in the virtual parallel machine.

• int rwapi_size ( ) returns the number of nodes in the virtual machine.

• void * rwapi_alloc ( size, net * ) allocates a contiguous memory block of the given size in the virtual address space of the process. If the underlying network interface requires the use of contiguous physical memory, it is attached to the application transparently. The value returned by this function is the virtual address in the virtual address space of the process where the contiguous memory block has been attached. The second parameter is the address where the “network” address will be stored when returning from the function. This address is the one that must be used for sending data.

• int rwapi_free ( void * ) deallocates the memory area provided as a parameter.

• int rwapi_send ( node, small ) sends a small message to another node. The value returned by this function is an error code.

• int rwapi_receive ( node *, small * ) returns information about the oldest incoming message that has not been taken into account yet. The value returned by the function is 0 if there is no message pending and 1 if a message has been taken into account. In this case, the node and the small message are stored at the addresses provided as parameters.

• sid rwapi_write ( node, net, net, size ) sends an arbitrary-long message to another. Both local and remote “network” addresses must be provided together with the size of the message. The Send ID (SID) returned by the function can be later used in order to determine if the message has been sent or not (this is useful to reuse a memory area).

• int rwapi_issent ( sid ) returns whether the message identified by the SID has been sent or not.

Note that, rwapi_ssend, rwapi_send, rwapi_issent and rwapi_receive are non blocking functions.

3 The Global Remote-Write Architecture

Even if minimalist, RWAPI is generic enough to be implement on top of any computer and communication architecture. Thus, in order to avoid rewriting several times the same pieces of code, we developed GRWA (the Global Remote-Write Architecture). GRWA is composed of a set of eight inter-dependable modules. Each module provides a specific set of services which can be used by the others.

Figure 2 presents the architecture of GRWA and the relations between the different modules (arrows on the figure represents the dependency between modules).

ARCH aims at providing a generic interface for processor specific functionalities. Typically, this includes the ability to read/write on I/O ports: almost all processors provide a way to let users access I/O ports; however, processor operations may vary from one processor to another.

BUS implements a separate module for bus management. This allows portability as some NICs are available for more than one I/O bus, and more than one NIC is available for a given I/O bus. It can also improve the efficiency of the interfaces provided by GRWA as it allows to replace the operating-system implementation by any customized ones.

CRYPT aims at ensuring the security of “network” addresses used to perform message transfers. At the physical layer, NICs are using physical addresses instead of virtual ones. A solution would consist in translating virtual addresses to physical addresses each time a transfer occurs. However, as whatever the way it is performed, the translation is costly, we consider it is better to provide users the addresses involved in message transfers. Then, provisions have to be made in order to check the validity of “network” addresses provided by the user [21, 19, 18].

INTERFACE provides users a unified API for message passing, as it is the only one the programmer should see. RWAPI is the default API provided by GRWA. However, other interfaces may be provided. For example, [12] shows it is possible to provide an efficient implementation of other
message-passing libraries (like MPI) using only the remote-write primitive.

**MEMORY** manages the memory used to transfer information between nodes. As memory areas used to send and receive data are not necessarily a multiple of a page size, and as memory is a critical resource (especially when the network interface requires the use of physical contiguous memory blocks), it is important to manage the memory resource carefully. As a result, this module manages physical and virtual memory undistinctively, in a similar way as the malloc-free couple of functions do.

**NETWORK** is in charge of managing NICs and routers. In order to improve performance, this management should be performed at the lowest level. This is the case for the implementation [20] on top of the Multi-PC machine [13] (using a HSL network [24]) which requires the application to deal directly with the NIC onboard component. However, it is possible to provide implementations on top of any other message-passing library, as long as the NETWORK module API is respected. This is useful to provide our interfaces on top of any new architecture until a native implementation becomes available.

**SYSTEM** aims at exporting data from kernel space to user space or at providing applications an access to functionalities which are traditionally reserved to privileged users.

**TOPOLOGY** provides a set of functions to determine which node number is attached to a hostname, the number of nodes in the parallel virtual machine and useful topological information to route messages.

## 4 InfiniBand

The InfiniBand Architecture (IBA) is a new industry-standard architecture for server I/O and inter-server communication. It was developed by the InfiniBand SM Trade Association (IBTA) to provide the levels of reliability, availability, performance, and scalability necessary for present and future server systems, levels significantly better than can be achieved with bus-oriented I/O structures.

InfiniBand Architecture is a message passing. It incorporates many of the concepts of the Virtual Interface Architecture. Each vendor had a different software stack with a proprietary value-add. However, there are multiple vendor-independent access layer that support different HCA (Host Channel Adapter, the network card) simultaneously called verbs. For example, the IBM InfiniBlue Host Channel Adapter Access Application Programming Interface is a vendor-independent interface that allows the development of both kernel-space and user-space applications based on the InfiniBand transport model. It has the ability to work with different verbs interfaces, and supports multiple channel adapters from the same or different vendors. It also recommends that the implementation ensure that multiple threads may safely use the APIs provided they do not access the same InfiniBand entity. Other verbs API include those defined by Mellanox and VIEO. The Mellanox IB-Verbs API (VAPI) interface provides a set of operations that closely parallel the proto verbs of the InfiniBand standard, plus additional extension functionality in the areas of enhanced memory management and adapter properties specifications. The VIEO InfiniBand Channel Abstraction Layer (CAL) Application Programming Interface provides a vendor-independent interface for InfiniBand channel adapter hardware. The CAL API evidently lies under the verbs abstraction, between the Channel Interface driver software and adapter hardware. It isolates specific hardware implementation details, providing both a common function call interface and a common data structure interface for the supported InfiniBand chipsets. An advantage of such an abstraction is the simultaneous support for heterogeneous channel adapters. This potentially enhances path selection.

In addition to the above interfaces that can be used to communicate with the Host Channel Adapters’ provider drivers directly, there also exist certain portable interfaces that hide the channel access interface from the user. These include the SRP, the IPoIB, the SDP, and the SM. The SRP (SCSI RDMA Protocol) enables access to remote storage devices across an InfiniBand fabric. The IPoIB provides standardized IP encapsulation over InfiniBand fabrics as defined by the IETF. This driver operates only within an InfiniBand fabric. The SDP (Sockets Direct Protocol) is an InfiniBand specific protocol defined by the Software Working Group (SWG) of the InfiniBand Trade Association (IBTA). It defines a standard wire protocol over IBA fabric to support user-mode stream sockets (SOCK_STREAM) networking over IBA. The SM (Subnet Manager) handles several areas related to the operation of the subnet including: discovery, monitoring and configuration of the ports connected to the subnet, responding to subnet administrative (SA) queries, configuration of I/O units with host channel drivers, performance management, and baseboard management.

There are many implementations of these components resulting in many software stacks. These stacks include the OpenIB which provides an open source implementation dedicated to linux 2.6 kernel, the IBAL (Intel Sourceforge InfiniBand project) which is the Intel open-sourced software stack that adapted to linux 2.4 kernels, the IBGD (IB Gold Distribution) stack is a full InfiniBand upper layer protocol...
stack for the Linux operating system hosted by Mellanox.

InfiniBand is an I/O channel based architecture rather than register based. A channel based architecture offloads the interprocessor communication overhead from the CPU to provide maximum available performance. For large multiprocessor applications the performance gain will be significant. The use of DMA engines at every InfiniBand node is critical to offloading the CPU. In a traditional load/store model, the data must pass through all levels of the memory hierarchy on its way to and from a CPU register. The device driver is responsible for maintaining the queue of transfers for the device. In an InfiniBand based system, I/O operations are scheduled as queued DMA operations and are handled by the node hardware rather than the CPU. The node hardware keeps track of which QPs have outstanding transfers, and completing those transfers. In addition to preserving memory bandwidth and precious cache resources, this architectural feature also minimizes the number of interrupts that must be serviced by the CPU during a data transfer.

The primary architectural element is the Queue Pair (QP). Each QP consists in an outbound queue, and an inbound queue. Queue Pairs are not shared between applications; therefore, once they are set up, they can be managed at the application level without incurring the overhead of system calls. The QP is the mechanism by which quality of service, system protection, error detection and response, and allowable services are defined. An application can use many QPs, each one with a different quality of service. Creating a QP requires support from the operating system which handles the HCA and initialize memory regions to be used by QPs to manage communication requests and memory operations.

The following transfer types are possible between Queue Pairs: Send includes a scatter/gather capability; RDMA-Write; RDMA-Read; Atomic Operations like Compare & Swap and Fetch & Add in remote memory; BIND WINDOW for remote memory management; and MULTICAST.

Each QP is configured for a particular type of service independently. These service types provide different levels of service and different error recovery characteristics. The available transport service types include: Reliable Connection, Unreliable Connection, Reliable Datagram, Unreliable Datagram and Raw.

Reliable Connection (RC) provides the highest level of reliability and predictability available today. In a RC the local QP is associated with exactly one remote QP forming a dedicated channel between the QPs. The hardware protocol between those QPs provides reliable transport. It detects missing, corrupted or invalid messages and automatically suspends further activity between the two QPs, resends the lost information, and then resumes operation. Every queue operation is acknowledged and every operation completes exactly once, in order. The maximum message size is not limited by packet size or by the maximum transfer unit (MTU) size defined for the channel. Segmentation and re-assembly of long messages happens in hardware and is transparent to the application. The Reliable Connection supports five InfiniBand services SEND, RDMA-Read, RDMA-Write, Atomic Operations, and BIND WINDOW.

Unreliable Connection (UC) also consists in a dedicated channel between local and remote QPs. However, in this case, there is no acknowledgement provided. Send queue operations are marked as complete as soon as the QP transmits them. A missed message or message received in error is not automatically retried, it is dropped and the QP does not provide the sender with any indication as to whether the message was successfully delivered or not. Like RC service, operations complete in order, and the maximum size for a message is not limited by the packet size or path MTU. SEND, RDMA-Write and BIND WINDOW are supported, but neither RDMA-Read, Atomic Operations nor MULTICAST are permitted over a UC channel. The UC provides more efficient communications, especially in streaming data applications, where a transient failure is not critical to output.

Unreliable Datagram (UD) service is connectionless. It has multiple receivers used during a scatter, gather operation to reliably determine data integrity as it goes to multiple target processor nodes. A QP can potentially send to any other UD QP in the system by specifying the remote QP number. Only SEND and MULTICAST are allowed. A QP configured for Unreliable Datagram service cannot detect messages delivered out of order nor can it detect messages delivered more than once. A message received in error will be dropped without notification. The maximum message size is limited by the largest packet size supported by the path MTU (256 bytes to 4 Kbytes). This is the same class of service provided by most existing LANs, where upper layer software protocols perform the transport and reliability functions.

Reliable Datagram (RD) service combines the features of both RC and UD services. Essentially, it provides a multiplexed reliable connection channel. RD combines the flexibility of multiple targets with the delivery guarantees of reliable connection. The QP is logically associated a set of remote RD QPs. Operations are acknowledged, complete exactly once, complete in order, and are automatically retried on error. All InfiniBand transfers are supported except the MULTICAST.
RAW service is used to send and receive packets for a protocol other than InfiniBand such as IPv6 or Ethernet packets. It supports only the SEND operation.

5 Performance measurements

In order to make the comparison between these communications libraries, we use three separate benchmarks (see Fig. 3).

The first benchmark (Fig. 3(a)) is the classic ping-pong in which a message can be sent only once the previous one has been received. The second one (Fig. 3(b)) is a bidirectional ping-pong which is used to highlight if a library takes benefits of the bidirectional links. The last benchmark (Fig. 3(c)) is the burst which aims at sending as many messages as possible regardless they have been received or not by the counter part.

Performance have been measured on Muse, one of the clusters of the Institut National des Télécommunications. Muse is composed of 16 nodes connected using both an InfiniBand 4× interconnection network for data and a Gigabit Ethernet interconnection network as a control network. Each node includes a 1.8-GHz AMD Opteron processor with a 1-MB cache and 2 GB of memory. For mass storage, each node is associated a 40-GB IDE hard drive, except for the first node (the front-end node) which is associated a 80-GB IDE hard drive; note that users accounts are stored on the front-end.

The measurement protocol is as follows: for each message size, each benchmark is run ten times. The duration of a run is one minute (this ensures a high consistency in results and we have determined that the confidence interval is greater than or equal to 90%). The system time is registered before the first message is sent (t₁) and after the last message is received on the same node (t₂). Let the elapsed time T be the difference between both. For a given run, let s be the size of messages and n be the number of effectively transmitted messages.

Let the end-to-end latency L (in the following we use the term latency) be the ratio between the elapsed time t and the number of effectively transmitted messages n. And let the user throughput T (in the following we use the term throughput) be the ratio between the amount of data (number of effectively transmitted messages n times the size of a message s) and the elapsed time t.

\[ t = t₂ - t₁ \]
\[ L = \frac{t}{n} \]
\[ T = \frac{n \times s}{t} \]

We compared our implementation with two other existing libraries on top of the InfiniBand Technology: VAPI, the native interface available on top of InfiniBand, and MPI developed on top of VAPI for InfiniBand.

Fig. 4 provides both latency and throughput for all three service types. These graphs shows that they all roughly provide the same communication performance. As a result, we chose to use RC only to make the comparison with the other two libraries (MPI and our remote-write message-passing interface RWAPI) as it is easy to work with and provides reliable communication.

Fig. 5 presents a comparison of performance for VAPI, MPI and RWAPI message-passing libraries. Fig. 5(c) and Fig. 5(f) show that RWAPI performance are very close to the native VAPI performance.

In a general way, Fig. 5 shows that RWAPI performance are always better than MPI performance. More specifically, the maximum ratio between the minimum latency achieved by RWAPI and the minimum latency achieved by MPI is up to 5.5 for small messages (i.e. 1.76 µs for RWAPI and 9.71 µs for MPI using the one-way benchmark).

Both RWAPI and MPI are able to achieve the maximum user throughput for long messages. However, RWAPI is able to provide this maximum user throughput for messages as short as 4 kilo-bytes while MPI cannot do the same for messages smaller than a few hundreds kilo-bytes.

Finally, the curves on Fig. 5 shows that there is an important difference in the management of short and long mes-
sages for MPI represented with a knee between 1 and 2 kilo-
bytes.

6 Conclusion and Future Works

In this paper, we have proposed a design of RWAPI over
InfiniBand. This design takes full advantages of the Infini-
Band hardware such as OS-Bypass and RDMA, thus elimi-
inating the involvement of the operating system and the re-
ceive process. In addition, it allows the overlap between
communication and computation.

To decrease the latency of small messages, we used the
Programmed-IO facility instead of RDMA to copy data to
the network card, so that removing one long-startup-time
DMA transaction.

Through performance evaluation, we have shown that
our design can achieve a low latency (about 1.76 µs) and
a high user throughput (more than 6300 Mb/s, i.e. the maxi-
nimum user bandwidth) even for short messages. As a com-
parison, the lowest latency provided by MPI over the same
platform is 4.96 µs and the maximum user throughput can-
not be achieved for messages smaller than several hundreds
of kilo-bytes.

Note that the lowest InfiniBand communication layer
(VAPI) let the user choose between producing events for
transfer operation completion or not. This do not suit
RWAPI as disabling events do not allow the user to be
informed about the completion of the send and enabling
events adds an extra overhead due to the unnecessary re-
ceive completion.

Currently, RWAPI uses RC as the type of service with
InfiniBand packet management. RC requires a connection
between each remote HCA and thus consumes much HCA
memory resources. Consumed memory is mainly used to
store data reassembly informations for each connection. To
achieve better scalability, we are working on applying the
RD type of service which bypasses any connection man-
age and maintains a reliable communication.

As a mid-term, we have planned to implement RWAPI
over Myrinet and Ethernet in order to perform communica-
tion over heterogeneous architectures composed of different
network types and different machine characteristics.

References

2.0.12, 1995.
[4] K. J. Barker. Runtime support for load balancing of paral-
Adviser-Nikos Chrisochoides.
system for pc++.
J. Selzovic, and W. Su. Myrinet – A Gigabit-per-Second
Mobile object layer: A runtime substrate for parallel adap-
tive and irregular computations. Advances in Engineering
ANL/MCS-TM-203, Mathematics and Computer Science
Division, Argonne National Laboratory, February 1995.
er IBM Document Number: SA22-7936-00 in IBM Reli-
able Scalable Cluster Technology for AIX L5. First Edition,
cation over MPI-1.
Figure 5. Performance comparison.