Remote-Write Communication Protocol for Clusters and Grids

Ouissem Ben Fredj
GET / INT
9, rue Charles Fourier
91011 Évry, France
Ouissem.BenFredj@int-evry.fr

Éric Renault
GET / INT
9, rue Charles Fourier
91011 Évry, France
Eric.Renault@int-evry.fr

ABSTRACT
Remote-write is a one-sided message-passing communication protocol adapted to cluster and grid computing. Its architecture is designed to make use of new features provided by recent Network Interface Cards such as Remote DMA and programmable card. This protocol offers a flexible programming model as well as a low overhead. This article is a survey of different programming models and their use. A second part presents a comparison of remote-write with other communication protocols according to the steps involved in the transfer’s critical path.

1. INTRODUCTION
In recent years, clusters of workstations have become a viable alternative to expensive supercomputers for high performance computing. With the introduction of high-speed networking hardware in LAN, the performance bottleneck in clusters was shifted from networking hardware to communication protocols and other machine resources.

Remote-write (RW) aims at proposing a communication protocol using new high-speed network features without complicating developer tasks. The resulting protocol uses a one-sided scheme as a programming model. This allows data to be moved from the sender to the receiver without any intervention of the receiver process.

RW is a protocol designed to exploit hardware and software resources to perform communications efficiently with a simple programming model. Its main features are:

- No synchronization needed between the sender and the receiver to perform a communication;
- Simple user-level interface;
- Possible removal of the operating system from the critical communication path (for user-level implementations);
- User-level memory management;
- Contiguous memory transfer to increase throughput.

The first three sections of the article are an introduction to the remote-write programming model. Section 2 is an overview of the one-sided scheme, section 3 describes the scheme and its use, and section 4 compares the remote-write programming model with other existing models. The second part provides insights into the design issues for the RW protocol with a comparison with other protocols. We concentrate on issues that determine the performance and the semantics of a communication system: memory management (section 5), host memory - NI data transfer (section 6), send and receive queue management (section 7), data transfer (section 8), and communication control (section 9).

2. OVERVIEW
In the literature, there are many communication protocols. Remote-write adopts the one-sided communication protocol for many reasons discussed in sections 3 and 4. A one-sided protocol is based on a single primitive, either a send or a receive primitive. A send-based protocol allows the sender to move its data directly to the receiver memory without intervention of the receiver process.

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 [1]. 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 is the thread-to-thread communication paradigm which is supported by CHANT [2]. The second is the remote service request (RSR) communication approach supported by libraries such NEXUS and DMCS. The third approach is a hybrid communication (that combine the two prior paradigms) supported by the TULIP [3] project. These cited paradigms are widely used. For example, Nexus supports the grid computing software infrastructure GLOBUS. MOL [4] extends DMCS with an object migration mechanism and a global namespace of the system. DMCS/MOL is used both in Parallel Runtime Environment for Multi-computer Applications (PREMA) [5] and in Dynamic Load Balancing Library (DLBL) [6].

In 1997, MPI-2 [7] (a new MPI [8] standard) have been including some basic one-sided functionalities. Although,
many studies have integrated one-sided communications to optimize MPI [9]. In 1999, a new communication library called Aggregate Remote Memory Copy Interface (ARMCl) [10] has been released. ARMCl is a high-level library designed for distributed array libraries and compiler run-time systems. IBM have maintained a low-level API named LAPI [11] implementing the one-sided protocol and running on IBM SP systems only. Similarly, Cray SHMEM [12] 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 [13] network uses the PCI-Direct Deposit Component (PCI-DDC) [14] to offers a message-passing multiprocessor architecture based on one-sided protocol. InfiniBand [15] proposes native one-sided communications. Myrinet [16, 17] and QNIX [18] do not provide native one-sided communications. But these features may be added (as for example in GM [19] with Myrinet since Myrinet NICs are programmable).

The arrival of this kind of networks has imposed common message-passing libraries to support RDMA (GM, VIA [20]. . . ). Note that 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.

Machine hardware have provided some kind of one-sided communication. Thinking machines [21] (CM1 in 1980, CM2 in 1988, and CM5 in 1992) are representative examples. CM5 uses two primitives, called PUT and GET, to allow thousands of simple processors to communicate over a Teraflops.

3. REMOTE-WRITE PROGRAMMING MODEL

R W uses the one-sided scheme as a programming model. It means that the completion of a send (resp. receive) operation does not require the intervention of the receiver (sender) process to take a complementary action. R W uses RDMA to copy data to (from) the remote user space directly. Figure 7? describes the different steps occurring in a basic communication between two processes. Suppose that the receiver process has 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 receive side, it keeps on for doing computation tasks, testing if new messages have arrived, or blocking until an incoming message event arises.

There are several classes of applications that are easier to write with one-sided communication:

- message passing algorithms;
- remote paging;
- adaptative codes where communication requirements are not known in advance;
- codes that perform random access to distributed data.

In this case the process owning the data does not know the data to access:

- asynchronous parallel algorithms;
- symmetric machines programming;
- data storage algorithms...

The RW programming model is simple, flexible and can be used as a high-level interface, or as a middleware between a high-level library such as MPI and the network level. A recent study proved that all MPI-2 routines can be implemented on top of a RW interface easily and efficiently [22]. Thus, any message-passing algorithms may be implemented using this programming model.

4. SYNCHRONIZATION

One way to compare communication libraries is to classify them according to the sender-receiver synchronization required to perform data exchanges. There are three synchronization modes: full synchronization mode, rendez-vous mode, and asynchronous mode.

With the full synchronization mode, sender have to ensure that the receiver is ready to receive incoming data. Thus, a flow control is required. FM [23] and FM/MC [24] implement flow control using a host-level credit scheme. Before a host sends a packet, it checks for credits regarding the receiver; a credit represents a packet buffer in the receiver’s memory. Credits can be handed out in advance by pre-allocating buffers for specific senders, but if a sender runs out of credits it must block until the receiver sends new credits.

LFC [25], specifically designed for Myrinet clusters, implements two levels of point-to-point synchronization: NI-level and host-level. At host level, when the NI (Network Interface) control program receives a network packet, it tries to fetch a receive descriptor. When the receive queue is empty, the control program defers the transfer until the queue is refilled by the host library. At NI-level, the protocol guarantees that no NI sends a packet to another NI before the receiving NI is able to store the packet. To achieve this, each NI assigns a number of its receive buffers to each NI in the system. An NI can transmit a packet to another one if there is at least one credit for the receiver. Each credit matches a free receive buffer for the sender. Once the sender has consumed all its credits for the receiver, it must wait until the receiver frees some of its receive buffers for this sender and returns new credits. Credits are returned by means of explicit acknowledgements or by piggybacking them on application-level return traffic. This mechanism is set up to all communication node’s pair, so that they are very expensive in NI memory resource and synchronization time. Indeed, for applications using a lot of small messages, NI buffers could overflow quickly and synchronization time may exceed the latency.

The rendez-vous mode discharge the duty of flow control to the application. For example, BIP [26], VIA [20], BDM [27] and GM require that a receive request is posted before the message enters the destination node. Otherwise, the message is dropped and/or NACKed. VMMC [28] uses a transfer redirection that consists in preallocating a default, redirectable receive buffer whenever a sender does not know the
The asynchronous mode breaks all synchronization constraints between sender and receiver. The completion of the send operation does not require the intervention of the receiver process to take a complementary action. This mode allows an overlapping between computation and communication, a zero-copy without synchronization, a deadlock avoidance, and an efficient use of the network (since messages do not block on switches waiting for the receive operation). As a consequence, the asynchronous mode provides a high throughput and low latency, in addition of a flexibility (as the synchronized mode can be implemented using the asynchronous mode).

AM [29] and PM2 (Parallel Multithreaded Machine) [30] are using the later mode to perform RPC-like communications. Each AM message contains the address of a user-level handler which is executed on message arrival with the message body as an argument. Unlike RPC, the role of the handler is to get the message out of the network and integrate it into the receiver process space. The problem with this scheme is that, for each message, a process handler is created (as with PM2) or an interrupt is generated (as with Genoa Active Message Machine (GAMMA) [31]) which is expensive (for both time and resource).

Some libraries (like VIA, AMII and DP [32]) require a startup connection to be executed before any communication. Such connection consists in creating a channel that allows communication between a sender and a receiver. This step is used most often to exchange capabilities (reliability level, quality of service...) and restrictions (maximum transfer size ...) of the process. This can be useful for dynamic and heterogeneous topologies.

As discussed previously, each synchronization mode has advantages and drawbacks. RW implements the asynchronous mode for both its simplicity and efficiency. The rest of the article analyzes each step of the communication’s critical path in order to implement the asynchronous mode efficiently.

5. MEMORY MANAGEMENT
Memory allocation precede any data transfer. It consists in reserving memory areas to store data to send or to receive. The way allocated areas are managed influences the performance.

Most of communication libraries use DMA to transfer data. The main constraint of DMA operations is that physical addresses are required rather virtual ones. Therefore, transfer routines must provide physical addresses of all pages of a message to the DMA engine. This is a tricky task because a contiguous message in the virtual address space is not necessarily contiguous in physical memory. A virtual-to-physical translation table built at allocation time can be used. Later, at the send (resp. receive) step, the translation table is used to gather (resp. scatter) data from (resp. to) memory to (resp. from) the network interface.

GM adds some optimizations to use the translation table: it stores the table in the user’s memory to be able to translate the whole memory and it creates a small cache table in the NIC memory. The cache table contains a virtual-to-physical translation of most used pages. To avoid page swapping, allocated buffer have to be locked.

Another solution used by the network layer of MPC-OS [33] consists in splitting the message to send into several smaller messages which size is less than the size of a page.

Yet another solution consists in managing physical addresses directly without operating system intervention. The idea is to allocate a physical contiguous buffer and to map it into the virtual contiguous address space. Thus, just one translation is needed. Its most important advantage is the avoidance of scatter/gather operations at transfer time. In FreeBSD, a kernel function allows to allocate physical contiguous buffers. In Linux, there are two methods to allocate physical contiguous memory. The first one is to change the kernel policy by changing the source code of Linux. The second one consists in allocating memory at boot-time. A driver maps the whole physical contiguous memory into a virtual contiguous area. Then, a function is used to search for a contiguous memory area that fits the requested size in the set of free spaces. Note that this function can be executed in user space without any call to the operating system.

Memory allocation is not a step of the communication’s critical path, but the policy used to manage memory has an important impact on data transfers. Our goal is to reduce the time spent in the virtual-to-physical translation by using physical contiguous memory allocations.

6. HOST MEMORY - NI DATA TRANSFER
With the RW protocol, the NI must communicate with the host memory in three cases. The first case is when the user process informs the NI for a new send. The user process sets up a send descriptor to be used by the NI to send message. Both the second and the third cases are when sending and receiving messages. For traditional message-passing systems, the user process must provide a receive descriptor to the NI. There are three methods to communicate between the host memory and the NI:

- PIO: The host processor writes (resp. reads) data to (resp. from) the I/O bus. However, only one or two words can be transferred at a time resulting in a lot of bus transactions. Throughput is different for write and read, mainly because writing across a bus is usually a little faster than reading;

- Write combining: It enhances write PIO performance by enabling a write buffer for uncached writes, so that affected data transfers can occur at cache line size instead of word size. Note that Write Combining is a hardware feature initially introduced on the Intel Pentium Pro and now available on recent AMD processors;
Choosing the suitable type of data transfer depends on the host CPU, the DMA engine, the transfer direction, and the packet size. A solution consists in classifying messages into three types: small messages, medium messages, and large messages. PIO suits small messages, write combining (when supported) suits medium messages, and DMA suits large messages. The definition of a medium message (and then the definition of both short and large messages) changes according to the CPU, the DMA engine, and the transfer direction. Since DMA-related operations (initialization, transfer, virtual-to-physical translation) can be done by the NI or the user process, a set of performance tests is the best way to define medium messages.

RW should take into account the strength of each method in order to communicate efficiently with the network interface.

7. SEND AND RECEIVE QUEUE MANAGEMENT

RW uses two queues to ensure data transfer: a send queue and a receive event queue. Unlike synchronous libraries, RW does not need a receive queue to specify receive buffers. Although, RW needs a receive event queue which contains a list of incoming messages.

Queues allow asynchronous operations. In fact, to send a message, the user process just appends a descriptor to the send queue. Once the operation is finished, the sender continues with the next send or with the computing task. Receive event queue is used to probe or poll for a new receive event.

The send queue contains a set of send descriptors provided by user processes and read by NI at send time. A send descriptor determines the outgoing buffer address, its size, the receiver node, the receiver buffer address, and the transfer type. Additional attributes can be specified to personalize the send (the security level, the reliability level). The NI uses send descriptors to initiate the send. Three steps are required to initiate a send.

The first one is the initialization of the send descriptor. This step is a part of the transfer’s critical path if the send is a point-to-point communication. For collective sends (multicast, broadcast...), this step can be done once for multiple send requests.

The second step consists in appending the send descriptor to the send queue. This step depends on the send queue management. In fact, according to the NI type and the memory size, the send queue can be stored either in the NI memory or in the host memory. The first case (used by FM and GM) avoids the NI from polling on host memory queue. The second case (used by the VIA specifications and implemented by Berkeley VIA [34]) allows a larger queue. MyVIA [35], an implementation of VIA over Myrinet, uses two queues (small ring in host memory and big ring on the NIC). If the small ring is not full, the send descriptor is written there directly. Otherwise, it is written in the big ring. If the number of unused descriptors in the small ring reaches a lower limit and if there are unprocessed descriptors in the big ring, the NI requests a driver agent to move big ring descriptors to the NI.

The third step of the critical path is the polling performed by the NI on the send queue. This step depends on the previous one. A comparison between MyVIA and Berkeley VIA proved that storing the send queue in the NI memory ensures a more efficient management of the send queue especially for small messages. In fact, Berkeley VIA requires two transactions between the NI and the host memory (the first are to inform the NI about the send and the second one to read the host memory descriptor) whereas MyVIA needs only one transaction. As for the send queue, the receive event queue should be stored on the host memory to allow easy polling by user processes.

Since the size of the send descriptor is only several byte long, PIO or write combining techniques should be used to update the NI send queue or to inform the NI about a new send.

8. DATA TRANSFER

As introduced in the section 3, RW is based on send function only. The receive operation consists in checking the receive event queue for receive operation completions. Receive operations are detailed in the next section.

To avoid bottlenecks and use available resources efficiently, a data transfer should take into account the message size, the host architecture (processor speed, PCI bus speed, DMA characteristics), NIC proprieties (transfer mode, memory size), and the network characteristics (routing policy, route dispersion...).

Many studies have tried to measure network traffics to check used message sizes. However, they mainly focused either on a set of applications [36], a set of protocols [37], a set of networks [38] or a specific environment (a single combination of network, protocol, machines and applications) [39]. All these studies show that small messages are prominent (about 80% less than 200 bytes). Moreover, RW requires an extra use of small messages to send receive buffer addresses. Thus, it is interesting for RW to distinguish between small and large messages. As discussed earlier, the maximum size of small messages should be determined using performance evaluation.

For the transfer of small messages, no send buffer address nor receive buffer address are required. Therefore, it is possible to store the content of small message in the send descriptor. To send such a message, as shown in Fig. 1, seven operations are performed: (1) the sender sets up the send descriptor (including data); (2) the sender informs the NI about the send descriptor; (3) the NI copies necessary data from the host memory, (4) the NI sends the message to the network; (5) the remote NI receives the message and appends it to the receive event queue; finally, (6) the receiver
The native model for message handling is polling. The networking the interrupt to the application in user space. The alternative approach, in which the kernel is involved in dispatching the arrival of a message by raising an interrupt. This is a driver-based communication system and a polling-based system. A network, a user may have to choose between using an interrupt-driven or polling-based system. When a message is detected, the polling function returns the receive descriptor describing the message.

Quantifying the difference in cost between using interrupts and polling is difficult because of the large number of parameters involved: hardware (cache sizes, register windows, network adapters), operating system (interrupt handling), run-time support (thread packages, communication interfaces), and application (polling policy, message arrival rate, communication patterns).

First, executing a single poll is typically much cheaper than taking an interrupt, because a poll executes entirely in user space without any context switching. Recent operating systems decrease the interrupt cost by saving minimal process state, but interrupt remain expensive. Second, comparing the cost of a single poll to the cost of a single interrupt does not provide a sufficient basis for statements about application performance. Each time a poll fails, the user program wastes a few cycles. Thus, coarse-grain parallel computing favors interrupts, while fine-grain parallelism favors polling.

For application containing unprotected critical sections, interrupts lead to nondeterministic bugs, while polling leads to safe run. Moreover, for asynchronous communication, polling can lead to substantial overhead if the frequency of arrivals is low enough that the vast majority of polls fail to find a message. With interrupts, overhead only occurs when there are arrivals.

Most of high-speed communication libraries (AM, FM, PM [40], GM) use polling and let interrupt to signal exceptions like queue overflow or invalid receive buffer address. FM/MC, LFC, and PANDA [41] uses a system that integrates automatically polling and interrupts through a thread scheduler. Since there is no incompatibility between RW and both polling and interrupt, users can use either polling or

9. COMMUNICATION CONTROL

The main focus of this section is how to retrieve messages from the network device. According to RW, NI informs the user process about the completion of the receive. When working on parallel machines with user-level access to the network, a user may have to choose between using an interrupt-driven communication system and a polling-based system.

Interrupt-driven approach lets the network device signals the arrival of a message by raising an interrupt. This is a familiar approach, in which the kernel is involved in dispatching the interrupt to the application in user space. The alternative model for message handling is polling. The network device does not actively interrupt the CPU, but merely sets some status bits to indicate the arrival of a message. The application is required to poll the device status regularly; when a message is detected, the polling function returns the receive descriptor describing the message.

Network policy can also affect data transfers. Adaptive routing, which allows multiple routes for each destination, may cause buffer overwriting due to unordered arrival of messages. This problem doesn’t exist with synchronous transfer.

Data transfer is the most important step of the communication. So care should be taken when writing its routines.

Figure 1: Short send with Remote-write.
interrupt depending on the application context. Note that, interrupts may be imposed by some libraries to guarantee a forward progress for system communication.

10. CONCLUSION
This paper presented several design issues for high-speed communication protocol. First, we classified protocols into three synchronization modes: full synchronization mode, rendezvous mode and asynchronous mode. Second, we detailed the different steps involved in the transfer’s critical path. Regarding the different implementation choices we draw the following conclusions:

- one-sided communications have been used for long;
- the use of either PIO, write combining or DMA for data transfer may be critical for efficient implementations;
- remote-write is suited to a large variety of applications;
- remote-write allows zero-copy transfer without synchronization;
- remote-write can take good advantage of physical contiguous memory;
- it is necessary to have messages which do not require addresses to be exchanged;
- no receive queue is required; however, the use of a receive event queue is preferred;
- there are no strong requirements regarding the use of polling or interrupts when receiving messages; however, the use of one against the other one is application dependent and performance results may be really different.

11. REFERENCES


[26] Loic Prylli and Bernard Tourancheau. BIP messages user manual.


