

# The Critical's Path of Communication Model Analysis for a Performant Implementation of High-speed Interfaces over the Myrinet Interconnect

Ouissem Ben Fredj and Éric Renault

Département Informatique

GET / INT

Évry, France

**Abstract** – *Recent high-speed networks provide new features such as DMA and programmable network cards. However standard network protocols, like TCP/IP, still consider a more classical network architecture usually adapted to the ethernet network. In order to use the high-speed networks efficiently, protocol implementors should use the new features provided by recent networks. This article provides an advanced study of both hardware and software requirements for high-speed network protocols. Results of the study has been used to design and implement the Remote-Write protocol over the Myrinet Interconnect. We obtained a very low latency and a throughput very close to the maximum user bandwidth for messages as small as 32 kB.*

**Keywords:** High performance computing, clusters, high-speed networks, communication protocol, performance measurement.

## 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 [9], Myrinet [4] and Quadrics. Two common features shared by these interconnects are User-level networking and Direct Memory Access (DMA). The Transmission Control Protocol (TCP) is one of the universally accepted transport layer protocols in today's networks. The introduction of high-speed networks a few years ago has challenged the traditional TCP/IP implementation in three aspects, namely performance, CPU requirements and memory traffic.

This paper is divided into two parts. First one analyses the existing communication models in order to choose the model that use new network hardware features.

The second part analyses the communication's critical path to determine the best way to take advantage of this new hardware. A comparison between existing high-speed communication protocols and libraries come with all steps of the study. Then, the Remote Write communication protocol is introduced. The last part analyses the performance of the implementation of the Remote Write protocol over the Myrinet network.

## 2 The communication models

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

With the full synchronization mode, the sender has to ensure that the receiver is ready to receive incoming data. This means that a flow control is required. FM [10] and FM/MC [12] both implement flow control using a host-level credit scheme. But if a sender runs out of credits it must block until the receiver sends new credits. This mechanism is set up to all communication node's pair, so that they are very expensive in both 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 [11], VIA [13], BDM [8] 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 [3] uses a *transfer redirection* that consists in pre-allocating a default redirectable receive buffer whenever a sender does not know the final receive buffer address. Later, when the receiver posts the

receive buffer, it copies the data from the default buffer to the receive buffer. To implement such a model, a middleware must be added between the user application and the implementation ensuring the flow control. Similar to VMMC, QNIX [14] program moves incoming data into a buffer in the NIC memory if incoming data arrive before the receiver creates the corresponding *Receiver Context*.

The asynchronous mode, also called the one-sided mode, breaks all synchronization constraints between senders and receivers. 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 a low latency, in addition to a flexibility (as the synchronized mode can be implemented using the asynchronous mode).

The one-sided 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 one-sided interface easily and efficiently [6]. This means that any message-passing algorithms may be implemented using this programming model. The one-sided scheme can be achieved either by using one-sided read or by a one-sided write. The remote read requires at least two messages, the first to inform the remote DMA engine (the remote network interface, the remote OS, or the remote process) about the requesting data and the second to send affectively the data. The remote write operation requires one message.

As discussed previously, each synchronization mode has advantages and drawbacks. The rest of the article focuses on the asynchronous mode for both its simplicity and efficiency.

### 3 Host memory - NI data transfer

According to the one-sided scheme, 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, ie, when 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, WC and DMA. With the Programmed IO (PIO), the host processor writes (resp. reads) data to (resp. from) the

I/O bus. This method is extremely fast. However, only one or two words can be transferred at a time resulting in lots of bus transactions. Throughput is different for writes and reads, mainly because writing across a bus is usually a little bit faster than reading. Write combining (WC) 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. Write Combining is a hardware feature initially introduced on the Intel Pentium Pro and now available on recent AMD processors;

A Direct Memory Access (DMA) engine can transfer entire packets in large bursts and proceed in parallel with host computation. Because a DMA engine works asynchronously, the host memory being the source or the destination of a DMA transfer must not be swapped out by the operating system. Some communication systems pin buffers before starting a DMA transfer. Moreover, DMA engines must know the physical addresses of the memory pages they access.

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. 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 (and then the definition of both short and large messages).

### 4 Data transfer

In order 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 properties (transfer mode, memory size), and the network characteristics (routing policy, route dispersion...).

Many studies have tried to measure network traffics to determine the size of messages. However, they mainly focused either on a set of applications [2], a set of protocols [5], a set of networks [15] or a specific environment (a single combination of networks, protocols, machines and applications) [7]. All these studies show that small messages are prominent (about 80% of the messages have a size smaller than 200 bytes). Moreover, the one-sided scheme requires an extra use of small messages to send receive buffer addresses. Thus, it is interesting 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, 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 process reads the data from the receive descriptor and (7) informs the NI that the receive is done successfully.

For the transfer of large messages, the sender has to specify both the send buffer address and the remote buffer address. Thus, there are two transactions added to the transfer. The first is to read the data from the host memory and the second is to write the data to its final destination.

It is a benefit to distinguish between small and large messages in order to use efficiently the hardware and to implement a performant one-sided scheme. Finally, data transfer is the most important step of the communication. So care should be taken when writing transfer's routine.

## 5 Communication control

The communication control focuses on how to retrieve messages from the network device. How should the NI inform the user process about the completion of the receive? With user-level access to the network, an implementor can choose between using interrupts or polling.

The interrupt-driven approach lets the network device signal the arrival of a message by raising up 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. In this case, 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.

Quantifying the difference 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), runtime 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 interrupts 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. Since there is no incompatibility between both polling and interrupt, users can use both depending on the application context. Note that, interrupts may be imposed by some libraries to guarantee a forward progress for system communications.

## 6 RWAPI

The previous study of both hardware and software requirements for high-speed network protocols has led to design the Remote Write protocol. Remote Write is one-sided communication protocol based on the remote write primitive. It lets the sender of a message provide all the information needed to copy a contiguous memory area from one node to another.

RWAPI (which stands for Remote-Write Application Programming Interface) is a lightweight interface designed to provide a lightweight 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 while requiring as less development as possible to port our interface to new architectures.

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; the size is 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 all 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. 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 )` checks whether the message identified by the SID has been sent or not.

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

## 7 RWAPI over Myrinet Interconnect

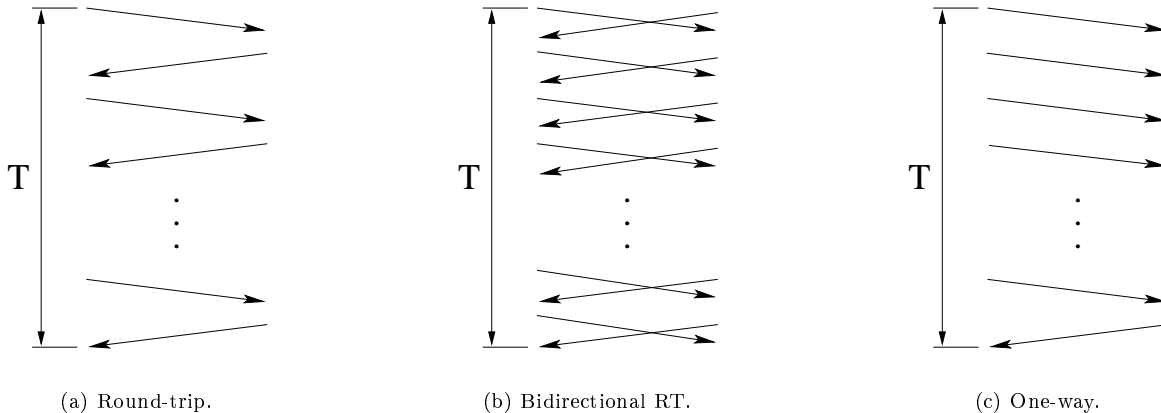
Myrinet [1], developed by Myricom, is a proprietary network technology and is compliant to the Physical

and Data Link layer defined in the ANSI/VITA 26-1998 standard. Myrinet is a switched, Gigabit per second network that is widely used in clusters. Myrinet adapters have a programmable network interface processor known as LANai. Several LANai version exist, the most recent are LANai9, LANaiXP, LANai2XP, and LANaiXM. Our testbed is equipped with LANai9 chip. LANai9 is a 32-bit RISC processor that operates at up to 133MHz for the PCI64B interfaces, or at up to 200MHz for the PCI64C interfaces. Using the Myrinet Control Program (MCP), which is stored in the onboard static RAM (SRAM), LANai9 controls the data transfer between the host and the network, performs data buffer management (through memory interface), and maintains network mapping and monitoring. The benefit of a programmable network processor is that it enables researchers to explore many protocol design options. Myricom recommend the GM API for both LANai9 and LANaiX and the MX API for LANaiX and LANai2X. Although, since the LANai processor is programmable, many message-passing libraries (like RWAPI, AM, FM, BIP, MyVIA) provide their own MCP to implement a specific network protocol.

There are many ways to implement the Remote-Write protocol on top of Myrinet. One solution would be a native implementation which would consist in developing a new MCP, a new kernel driver and a new user library. This solution is under development and good performance are expected. However, this would not be portable since an MCP should be provided for every network card version. To overcome this limitation, we developed the Remote-Write protocol on top of GM. This way, RWAPI is automatically available for all architectures and NIC versions that GM supports.

In order to launch application’s processes according to the SPMD model, we use an SSH-based spawner which creates a *master process* on the current host and one process on each host provided as an argument. Then, each process can communicate with the *master* to get information such as the process rank, the number of processes, the application’s arguments and other control informations. Then, processes can perform collective operations on top of socket-based connections. All collective operations are used only to initialize and finalize the application transparently to the user.

In order to associate a process rank to its GM ID, processes perform an exchange of GM IDs using the *exchange* operation set up by the *master process*. To maintain the non-blocking semantic of RWAPI operations (`rwapi_send`, `rwapi_write`, `rwapi_receive`, and `rwapi_issent`), we used a host memory receive list (HMRL). Thus, when a receive message event arises while the process is not waiting for messages, the interface copies the content of the message event in the HMRL. Note that message events do not contain user



**Figure 1. Performance benchmarks.**

data except those corresponding to `rwapi_send` operations. In this case, the length of the user message is limited to 128 bits and thus a copy of this message is not expensive.

The `rwapi_send` function insures that GM is ready to send messages before calling the `gm_send_to_peer_with_callback` function. The later function is the fastest send function provided by GM since it uses the same port number as the sender at the receiver side. We set up GM to copy the data in the send descriptor to avoid an expensive DMA operation performed by the MCP to copy data from the host memory to the NIC memory. We also use Write-Combining feature instead of PIO or DMA to copy the send descriptor from the host memory to the NIC memory. This feature combines many PIO operations and is adapted to medium message size. Similar to the `rwapi_send` function, the `rwapi_write` function insures that GM is ready to send messages before calling the `gm_put` function. The `gm_put` function is suited to the remote-write paradigm since it guarantees the minimum number of copies while transferring the data. Indeed, it copies the data from the host memory directly to the NIC memory without any system call.

## 8 Performance analysis

We compared our implementation with the version of MPI, the other existing library available for the Myrinet-2000 Technology, developed on top of GM. For both MPI and RWAPI we developed our own benchmarks as presented above. We use three separate benchmarks (see Fig. 1). The first benchmark (Fig. 1(a)) is the classic ping-pong in which a message can only be sent once the previous one has been received. The second one (Fig. 1(b)) is a bidirectional ping-pong which is used to highlight the capability of a library to take benefits of bidirectional links. The

last benchmark (Fig. 1(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 POETS, one of the clusters of the Institut National des Télécommunications. POETS is composed of eight nodes connected using both a Myrinet interconnection network for data and a Gigabit Ethernet interconnect as a control network. Myrinet adaptors are 133-MHz LANai-9 (M3S-PCI64B) with a PCI connector. The host adaptor is equipped with 2 MB of memory. The firmware used for testing was GM 2.0.9. Apart from GM, the port of MPICH on top of GM called MPICH-GM version 1.2.6..14b for Linux x86 was also installed. Each node includes a 800-MHz Intel Pentium III processor with 1.2 GB of memory. The front-end of the cluster is an extra node with almost the same characteristics except that it includes no Myrinet NIC.

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 all confidence intervals are greater than or equal to 90%). The system time is registered before the first message is sent ( $t_1$ ) and after the last message is received on the same node ( $t_2$ ). Let the elapsed time  $t$  be the time 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 times the size of a message) and the elapsed time.

$$t = t_2 - t_1 \quad L = \frac{t}{n} \quad T = \frac{n \times s}{t}$$

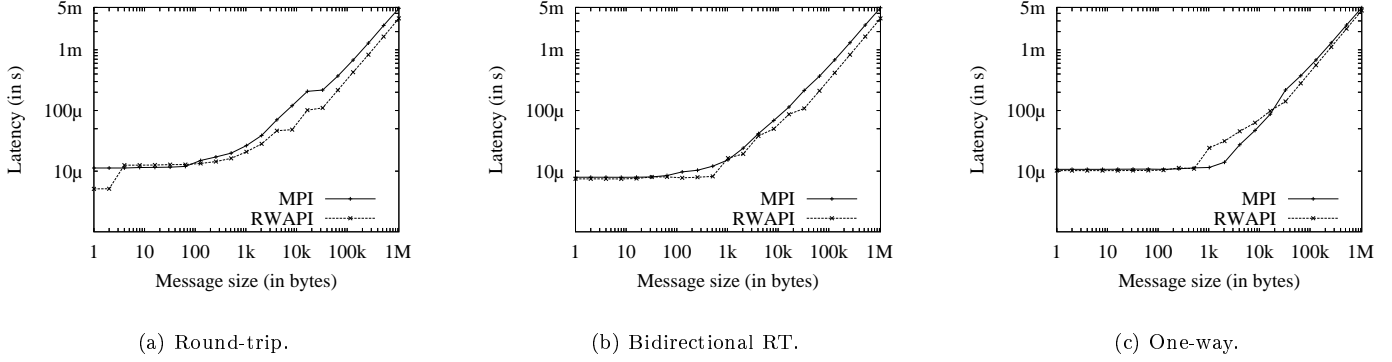


Figure 2. Latency.

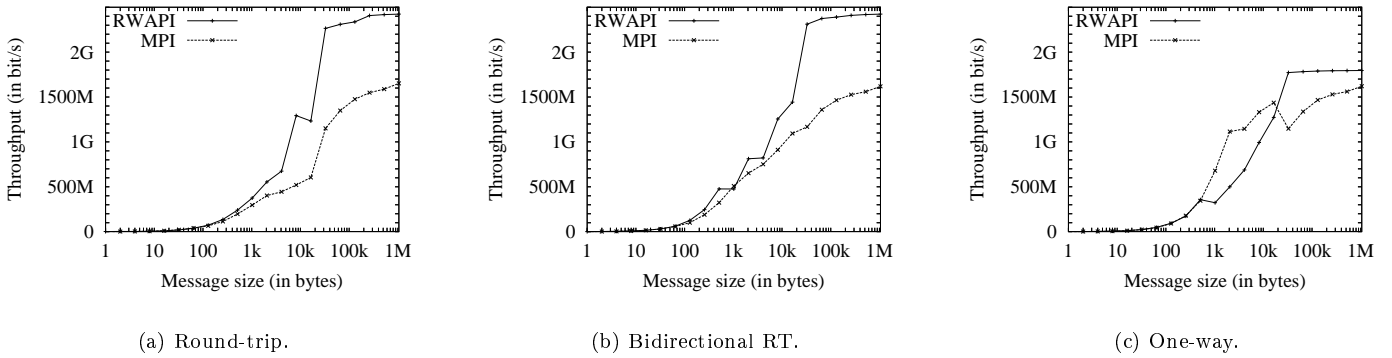


Figure 3. Throughput.

Since the performance for both RWAPI and MPI are almost the same for messages size greater than or equal to 1 MB, we do not include data for larger messages.

Fig. 2 presents the latency of RWAPI and MPI for various message sizes. Note that for readability reasons, a logarithmic scale have been used for latency curves. All these graphs highlight that, in the general case, performance with RWAPI are usually better than those with MPI. Exceptions are for the One-Way benchmark for messages which size ranges from 1 kB to 16 kB and for the Round-Trip benchmark for messages which size ranges from 4 bytes to 64 bytes. This may be due to the fact that MPI is using Write-Combining to copy data from the host memory to the NIC memory for small messages avoiding the overhead of the DMA start-up. Regarding the latency, an interesting result is that for the Round-Trip benchmark (see Fig. 2(a)), the minimum latency for RWAPI is as low as  $5.1 \mu\text{s}$ . As a comparison, the minimum latency for MPI is  $7.9 \mu\text{s}$  and is achieved using the Bidirectional Round-Trip (see Fig. 2(b)).

Fig. 3 shows the throughput of RWAPI and MPI for various message sizes. All these graphs highlight that, in the general case, performance with RWAPI are usu-

ally better than those with MPI.

These graphs are highlighting three very important results. First, RWAPI is able to achieve a maximum throughput of up to 2.43 Gb/s for large messages (see Fig. 3(a) and Fig. 3(b) for Round-Trip and Bidirectional Round-Trip benchmarks respectively). As a comparison, the maximum throughput provided by MPI is 1.66 Gb/s. This means that MPI cannot offer more than 68.4% of the maximum throughput offered by RWAPI. In other words, RWAPI is able to deliver large messages 46.3% faster than MPI (in fact, this is not highlighted with latency graphs on Fig. 2 as a logarithmic scale is used for messages larger than 1 kB).

Finally, RWAPI provides a larger part of the bandwidth for smaller messages than MPI. Typically, RWAPI is able to achieve 90% of the maximum throughput for 32-kB messages as MPI requires messages of 256 kB to do the same.

## 9 Conclusion

This paper presented several design issues for high-speed communication protocol. First, we classi-

fied communication models into three synchronization modes: full synchronization mode, rendez-vous mode and asynchronous mode. The asynchronous mode removes all synchronization constraints between the sender and the receiver. Moreover, this mode is suited with one-sided programming model which offers a simple programming interface and high performance. In addition, both the asynchronous mode and the one-sided scheme take advantage of the DMA feature to achieve a RDMA communication. Note that the one-sided programming model insures the implementation of all message-passing application. Finally, the RDMA-write is sufficient to implement both these two modes.

To achieve a good communication control, the programmer should use both interruption and polling. Polling is suited with fine-grain applications and interrupt is used with coarse-grain one. The programmer should avoid all unprotected critical section and should use the interrupt to signal exceptions such as the overflow of the buffer or a security problem. The communication routines should take into account that a big number of small message may be exchanged. These messages correspond to memory address which are used in the RDMA communication. Thus, a distinction between small message and large message is interesting. For small message, PIO or write-combining method is usable and DMA is suitable for large message.

The previous study led to design our own communication protocol called Remote Write. The second part of this paper, we have presented the design and implementation of RWAPI over Myrinet-2000. This design takes full advantages of the Myrinet-2000 hardware such as OS-Bypass and RDMA, thus eliminating the involvement of the operating system and the receive process. In addition, it allows the overlap between communications and computations.

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 of 5.1  $\mu$ s and a high user throughput of 2.43 Gb/s even for relatively short messages (2.26 Gb/s is available for 32-kB messages). On the same platform, the lowest latency provided by MPI is 7.9  $\mu$ s and the maximum user throughput provided by MPI represents 68.4% of the maximum user throughput provided by RWAPI (this means that message transfer with RWAPI is up to 46.3% faster than with MPI).

As a short-term, we have planned to optimize the `rwapi_write` operation by using either PIO, Write-Combining or DMA to copy user data from the host memory to the NIC memory; PIO operations for very small messages, Write-combining for medium messages

and DMA for large messages since it involves an expensive start-up overhead.

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

## References

- [1] ANSI/VITA 26-1998. Myrinet-on-VME Protocol Specification., 1998.
- [2] N. Basil and C. Williamson. Network traffic measurement of the x window system.
- [3] M. A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. W. Felten, and J. Sandberg. Virtual memory mapped network interface for the shrimp multicomputer. In *ISCA '98: 25 years of the international symposia on Computer architecture (selected papers)*, pages 473–484. ACM Press, 1998.
- [4] N. Boden, D. Cohen, R. Flederman, A. Kulawik, C. Seitz, J. Selzovic, and W. Su. Myrinet – A Gigabit-per-Second Local-Area Network. volume 15, pages 29–36, 1995.
- [5] J. Cao, W. S. Cleveland, D. Lin, and D. X. Sun. On the nonstationarity of internet traffic. In *SIGMETRICS '01: Proceedings of the 2001 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 102–112. ACM Press, 2001.
- [6] O. Glück. *Optimisations de la bibliothèque de communication MPI pour machines parallèles de type << grappe de PCs >> sur une primitive d'écriture distante*. PhD thesis, Université Paris VI, July 2002.
- [7] R. Gusell. A measurement study of diskless workstation traffic on an ethernet. In *IEEE Transaction on Communications*, volume 38, pages 1557–1568, September 1990.
- [8] G. Henley, N. Doss, T. McMahon, and A. Skjellum. Bdm: A multiprotocol myrinet control program and host application programmer interface, 1997.
- [9] InfiniBand Trade Association. *The InfiniBand Architecture, Specification Volume 1 & 2*, June 2001. Release 1.0.a.
- [10] S. Pakin, M. Lauria, and A. Chien. High performance messaging on workstations: Illinois Fast Messages (FM) for Myrinet. pages ??–??, 1995.
- [11] L. Prylli and B. Tourancheau. BIP messages user manual.
- [12] K. Verstoep, K. Langendoen, and H. E. Bal. Efficient reliable multicast on myrinet. In *ICPP'96: Proceedings of the Proceedings of the 1996 International Conference on Parallel Processing, Vol. 3*, pages 156–165. IEEE Computer Society, 1996.
- [13] Virtual Interface Architecture Specification, Version 1.0, published by Compaq, Intel, and Microsoft. December 1997.
- [14] A. D. Vivo. *A Light-Weight Communication System for a High Performance System Area Network*. PhD thesis, Università di Salerno - Italy, November 2001.
- [15] C. Williamson. Internet traffic measurement. *IEEE Internet Computing*, 5(6):70–74, 2001.