Efficient Communications in Multithreaded Runtime Systems

Luc Bougé, Jean-François Méhaut, and Raymond Namyst

Laboratoire de l'Informatique du Parallélisme, École normale supérieure de Lyon, F-69364 Lyon, France.
{Luc.Bouge,Jean-Francois.Mehaut,Raymond.Namyst}@ens-lyon.fr

Abstract. Most of existing multithreaded environments have an implementation built on top of standard communication interfaces such as MPI which ensures a high level of portability. However, such interfaces do not meet the efficiency needs of RPC-like communications which are extensively used in multithreaded environments. We propose a new portable and efficient communication interface for RPC-based multithreaded environments, called MADELEINE. We describe its programming interface and its implementation on top of low-level network protocols such as VIA. We also report performance results that demonstrate the efficiency of our approach.

1 Introduction

For portability reasons, most existing multithreaded environments are often implemented on top of high-level standard communication interfaces such as PVM or MPI. A large subset of these environments are RPC-based environments because the provided functionalities rely implicitly or explicitly on a mechanism able to invoque remote services (*e.g.* RSR in Nexus [7], RPC in PM2 [12], Fork/Join in DTS [4]).

In fact existing communication interfaces do not adequately support the implementation of these environments [2], either because they are not portable (too low-level) or simply because they do not match the specific needs of these environments. Usually, implementations fall into the second category because most of existing environments emphasize portability (*e.g.* Chant [9], Athapascan [3]).

This situation, often been considered as a "natural" tradeoff between portability and efficiency, can be avoided. In this paper, we present a new portable communication interface (called MADELEINE) that provides both efficiency and high-level abstractions to RPC-based multithreaded environments. The software structure of MADELEINE is organized in two layers. The low-level layer (portability) isolates the code that needs to be adapted for each targeted network protocol. The high-level layer (API) provides advanced generic communication functionalities to optimize RPC operations.

MADELEINE is easily portable on low-level network protocols and is currently available on top of VIA [5], BIP [14], SCI [6], SBP [15], TCP, PVM [8] and MPI

[11]. Only a small constant overhead is introduced by MADELEINE on top of all these underlying procotols. Thus, its performance competes with low-level communications packages such as FM [13] or AM [16].

The remainder of this article is organized as follows. In the next section, we introduce the specific needs of RPC-based MTE as far as efficiency is concerned. The contribution of our work is presented in Section 3 where the MADELEINE communication interface is described and discussed. Section 4 describes the implementation of MADELEINE on top of several network protocols and shows some performance results. Section 5 presents related work. Finally, Section 6 summarizes the contribution of this work and lists the points we intend to address in the near future.

2 Zero-copy data transmissions in RPC-based Multithreaded Environments

A multithreaded environment is called "RPC-based" if most communications take place by means of remote invocations of services (also called Remote Procedure Calls). In this scheme of remote interaction, a client (usually a thread) sends a request to a server (usually a process) which executes a specified function (a service) to serve the request. According to the type of service executed, a response may be sent back to the client upon completion of the function. Many kinds of remote operations actually conform to this communication scheme and one can observe that most existing multithreaded environments provide RPC-based mechanisms [7, 9, 3, 4].

On a high speed network providing very low transmission latency (*e.g.* Myrinet [1]), any extra message copy introduced at some software level has a tremendous impact on performance [10]. Therefore, it is crucial that the underlying communication layer be able to ensure the delivery of messages with no intermediate extra copy of data. Many existing communication libraries actually provide such a functionality. However, the problem of realizing an RPC operation without any extra copy of data is more complex than realizing a zero-copy message transmission. Actually, when a RPC request is sent to a server, the server does not know the location where the data should be placed until it extracts preliminary information from the request message. Typically, the triggering of a RPC operation takes place in the following three steps on the server side:

1. The request type is identified (*i.e.* "Is this a migration or a remote put?") and some information is extracted from the network.
2. Then, actions are taken (for instance dynamic memory allocations) to prepare for the receipt of the data.
3. Finally, the data are extracted from the network and stored directly at the right location in memory.

As a result, implementing this scheme while avoiding extra copies of messages will require the client to send two messages (at least with a classical message passing interface): the first one to carry the "*header*" of the request and the second one to carry the regular data (the "*body*"). However, this approach does not yield good results if the underlying communication software is still buffering data on the receiving side. In this case, a single message would probably have been sufficient to carry all the information (header + body), since it could have been extracted in several steps.

3 The MADELEINE Communication Interface

To meet the efficieny and portability requirements of RPC-based multithreaded environments, we have designed a new communication interface called MADELEINE. Unlike environments like MPI or PVM, MADELEINE is not designed to be used directly by regular user-level applications. This interface is especially targeted at RPC-based multithreaded environments such as PM^2, Nexus or Chant. In this context, it is intended to bridge the gap between functionalities provided by low-level network protocols (such as BIP [14] or VIA [5]) and requirements of high-level abstractions (such as the RPC mechanism).

One important feature of MADELEINE is that it was designed to allow upper software layers (*i.e.* inside the multithreaded environment) to avoid extra copies of transmitted data. At the lowest level, this cooperation is realized using up-calls on the receiving side to allow upper layers to interactively participate in data emissions and extractions. At the user interface level, this cooperation is elegantly accomplished through the use of simple buffer management primitives.

Another main characteristic of MADELEINE is its "thread-awareness", which means that several threads may access its interface concurrently. Clearly, adequate synchronisation is enforced wherever necessary. For instance, only one thread is allowed to transmit data on a given communication link (in a given direction) at a time. Moreover, MADELEINE is able to properly schedule threads accessing to the network. For instance, polling operations may be grouped in order to avoid multiple threads simultaneously busy waiting on the same network interface.

The structure of MADELEINE is actually composed of two software layers: a generic user programming interface and a low-level portability interface whose implementation is network-dependent.

3.1 The Programming Interface

The MADELEINE programming interface provides a small set of primitives to allow the building of RPC-like communication schemes. Theses primitives look like classical message-passing-oriented primitives, and they actually only differ in the way the reception of data is handled. Basically, this interface provides primitives to send and receive *messages*, and several *packing* and *unpacking* primitives that allow the user to specify how data should be inserted into/extracted from messages.

Messages may consist of several pieces of data, located anywhere in userspace. They are assembled (resp. disassembled) incrementaly using *packing* (resp. *unpacking*) primitives. This way, a message can be built (resp. examined) at multiple software levels, which allows for instance the use of piggybacking techniques without losing efficiency. The following example illustrates this need. Let us consider a remote procedure call which takes an array of unpredictable size as a parameter. When the request reaches the destination node, the header is examined both by the multithreaded runtime (to allocate the appropriate thread stack and then to spawn the server thread) and by the user application (to allocate the memory where the array should be stored). This shows that several software layers may actually participate in the packing/unpacking of message data.

Sending messages Let us now describe how a user program prepares and sends a message to a remote node (Fig. 1). Such an operation starts by asking Madeleine to allocate a new "message descriptor" for a given destination. Then, a series of *packing* operations are performed to "register" the differents part of the message. Finally, the emission operation is requested.

```
sendbuf_init(dest_node);

   sendbuf_pack ...
   sendbuf_pack ...

sendbuf_send();
```

Fig. 1. A typical send operation.

```
void handler()
{
  recvbuf_unpack ...
  recvbuf_unpack ...

  recvbuf_receive_data();
}

recvbuf_receive(handler);
```

Fig. 2. A typical receive operation.

The critical point of a send operation is obviously the series of *packing* calls. Such packing operations simply *virtually* append some data to a message under construction. Thus, data may not be copied, but just registered as part of the message.

The prototypes of these functions are very similar to the buffer management primitives provided by the PVM interface (*e.g.* `pvm_pkint`, `pvm_pkfloat`, etc.). For instance, the function used to append one or more integers to the current message has the following prototype:

```
void sendbuf_pack_int(int mode, int *elems, int nb);
```

Among the parameters, `elems` is the address of an array of integers and `nb` represents the size of this array. The `mode` parameter plays an important role in the Madeleine interface, because it determines the semantics of the operation.

It can be assigned different values, also called flags. These flags, defining the semantics of the `sendbuf_pack` operation, are mutually exclusive:

`SND_SAFER` When used, this flag indicates that MADELEINE should pack the data in a way that further modifications to the corresponding memory area should not impact on the message. In other words, an explicit "copy" is required. This is particularly mandatory if the data location is reused before the message is actually sent.

`SND_LATER` This flag indicates that MADELEINE should not consider accessing (*i.e.* copying or even sending) the value of the corresponding data until the `sendbuf_send` primitive is called. This means that any modification of these data between their packing and their sending will actually update the message contents. This is typically useful when a message is to be sent to various destinations with only minor variations. In this case, the data can be packed only once and their values can be updated between the send operations.

`SND_CHEAPER` This is the default flag. It allow MADELEINE to do its best to handle the data as efficiently as possible. The counterpart is that no assumption should be made about the way (and the time) MADELEINE will access the data. Thus, the corresponding data should be left unchanged until the send operation has completed. We describe in Section 4 how this can be implemented according to the underlying network protocol. Note that most part of data transmissions involved in parallel applications can accomodate the `SND_CHEAPER` semantics.

Of course, packing the same data multiple times with different semantics within the same message produces an erroneous program. There is no other particular restriction on the order or on the number of packing operations with respects to their semantics. The only constraint imposed by MADELEINE is that the sequence of packing operations will have to be "mirrored" on the receiving side, as seen in the next section.

Receiving messages Receiving a message is always done in three steps (Figure 2). First, the arrival of the message is detected (inside `recvbuf_receive`), which triggers the execution of a handler. Second, a number of unpacking operations (`recvbuf_unpack_*`) are performed within the handler to specify where data should be stored in memory. Third, a "commit" operation is executed (`recvbuf_receive_data`) to force the completion of the data transmission.

The last two steps need further explanations. As the *packing* operations, the *unpacking* ones also require a special parameter which specifies *when* the data should be received. There are two possible values for this parameter:

`RECV_EXPRESS` This flag forces MADELEINE to guarantee that the corresponding data is immediately available upon the completion of the *unpacking* operation. Typically, this flag is mandatory when extracting data that will be used before the `recvbuf_receive_data` primitive is called. On some network protocols, this functionality may be available for free. On some others, it could penalize latency and bandwidth. That for, the user should extract data this way only when necessary. In the example described in section 2, the message "*header*" would have to be unpacked this way.

`RECV_CHEAPER` This is the default flag. It allows MADELEINE to defer the extraction of the corresponding data until the execution of `recvbuf_receive_data`. Thus, no assumption can be made about the exact moment at which the data will be extracted. Depending on the underlying network protocol, MADELEINE will do its best to minimize the overall message transmission time.
If combined with `SND_CHEAPER`, this flag always guarantees that the corresponding data is transmitted as efficiently as possible. On high-performance network protocols such as VIA or BIP for instance, data will effectively be transmitted without any extra copy.

For efficiency reasons, the way some data is to be extracted from a message should be known by MADELEINE on the sending side. That is, when some data is packed into a message, the programmer has to specify not only the sending mode, but also the receiving mode (for clarity, we omitted this detail in the previous section). Furthermore, the reverse is also true for the same reason: when unpacking some data out of a buffer, the sending mode has to be specified in addition to the receiving mode (by logically *or*-combining the flags). This constraint could have been avoided at the price of a significant performance drop (messages would need to be self-describing), so we did not retain this solution. Because MADELEINE internal messages are not self-describing, *packing* and *unpacking* operations have to be done in the same order (matching pack and unpack operations must have the same rank within each series).

Illustration We now describe a small example which illustrates the powerfulness of the MADELEINE interface. Let us say that we want to send a message containing two strings of arbitrary length (unpredictable on the receiving side). These strings are allocated by a function (`generate_str`) returning their address in memory. On the receiving side, it is necessary to get the size of each string (an integer) before extract the string themselves (destination memory has to be dynamically allocated). So the message will contain four elements (two integers and two strings) with a first constraint: each integer must be extracted `RECV_EXPRESS` before the corresponding string.

On the sending side, the code may look like the one in Figure 3. Note that the integers are packed using the `SND_SAFER` mode because the `size` variable is reused inside the loop.

Figure 4 shows the corresponding receiving code. Using the `RECV_CHEAPER` mode for the transmission of strings may allow MADELEINE to send both objects without extra copies in a single network transaction (with a gather/scatter mechanism). This is a major advantage over communication interfaces such as FM [13] that do not allow the user to express such a semantics. Thus, either a extra copy or an extra network packet would have to be sent in such a case.

```
char *data;
int i, size;

   sendbuf_init(dest_node);

   for(i=0; i<2; i++) {
      data = generate_str(); size = strlen(data)+1;
      sendbuf_pack_int(SND_SAFER | RECV_EXPRESS, &size, 1);
      sendbuf_pack_byte(SND_CHEAPER | RECV_CHEAPER, data, size);
   }

   sendbuf_send();
```

Fig. 3. Sending a sample message with the MADELEINE Programming Interface

3.2 The Portability Layer

The *Portability Layer* of MADELEINE is intended to provide a common interface to the various communication subsystems on which it may be ported. The design of this interface is critical because it has to be independent of any particular protocol while staying efficiently portable on top of any of them.

We chose an execution model that is inspired by the Active Messages model [16]. Thus, our model is essentially a message-passing protocol where exchanges are done between traditional processes. This means that although its implementation has to be thread-safe, the protocol does not need to be thread-aware. As a consequence, upper layers have to ensure that only one thread is processing a receive operation at a time.

Messages consist of a set of one or more vectors. Each vector is a contiguous area of memory and is defined by a pair (*address*, *size*) where the size is expressed in bytes. The first vector of a message has a particular semantics with respect to the receive operation: this vector contains the data that must be extracted prior to the rest of the message (*i.e.* the “header”). Once extracted, these data are immediately made available to the upper layers so that some application-level actions may be executed before the rest of the message is extracted.

The Figure 5 sketches a situation where process A is sending a message to process B. First, the message *header* (*i.e.* the first vector) is sent to process B. On its receipt, a *handler* is executed with the header as a parameter. This handler, which is defined by the upper layer, can inspect the header and take appropriate actions to prepare for full message extraction. Typically, the handler may compute the locations where data should be placed (step (3) in Figure 5). Process A is allowed to send the vectors of remaining data (i.e. the message body) when the handler has completed. Consequently, these data are directly stored at the right memory locations on their receipt.

We emphasize that this scenario is a conceptual description of the protocol. In fact, specific implementations may not strictly follow this communication scheme. We discuss more precisely the implementation issues in Section 4.

```
char *data[2];

void handler()
{
   int i, size;

   for(i=0; i<2; i++) {
      recvbuf_unpack_int(RECV_EXPRESS | SND_SAFER, &size, 1);
                           /* ‘size’ is available immediately */
      data[i] = malloc(size);
      recvbuf_unpack_byte(RECV_CHEAPER | SND_CHEAPER, data[i], size);
                   /* Caution: data is *not* yet available here. */
   }
   recvbuf_receive_data();
}

int main_function()
{
   recvbuf_receive(handler); /* The current thread will block and  */
                             /* wait for the arrival of a message. */
   printf("I received the strings: %s and %s\n", data[0], data[1]);
}
```

Fig. 4. Receiving and inspecting a message with the MADELEINE Programming Interface

The Portability Interface The portability interface we propose is composed of only 8 function prototypes (Table 1) which represent the architecture-dependent primitives of MADELEINE. Two of them (`init` and `exit`) deal with the management of connections respectively at the beginning and at the end of the execution of an application. The remaining 6 primitives deal with communication.

Message sending As MADELEINE is intended to be used in a multithreaded context, operations that would ordinarily block the calling process (*e.g.* the sending of a message) should only block the calling thread. However, the portability layer has no knowledge about the thread package which is actually used in the upper layers.

Therefore, a sending operation is done in two steps. A call to the `send_post` primitive initiates the sending of a message and returns without blocking. The destination process and the message (an array of Unix standard `iovec` structures) are given as parameters. Then, further calls to `send_poll` have to be made until the primitive returns TRUE, which means that message transmission has completed on the sending side.

Message receiving During the receipt of a message, the first step is to receive the message header, which is done by a call to `recv_header_post(handler)`

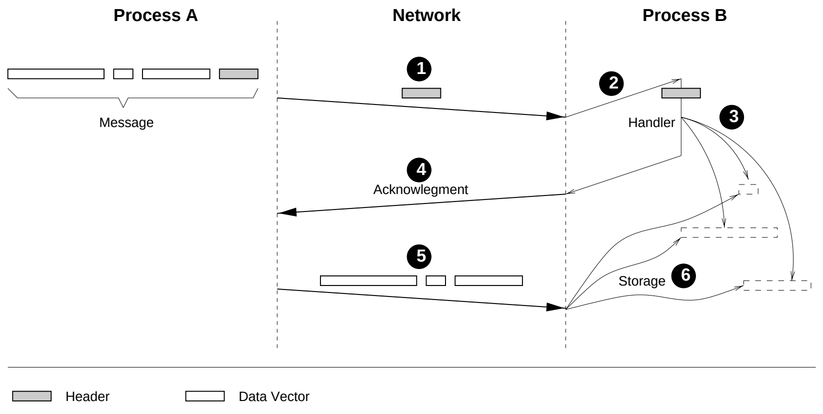

Fig. 5. Conceptual view of the data-exchange protocol in the portability layer.

Table 1. The MADELEINE portability interface.

Function	Operation
`init(configSize, taskIDs)`	Connections Setup
`exit()`	Connections Shutdown
`send_post(dest, iovec, count)`	Post the sending of msg to node 'dest'
`send_poll(dest)`	Check if msg sending is completed
`recv_header_post(func)`	Ready to receive a header
`recv_header_poll()`	Check if header received
`recv_body_post(exped, iovec, count)`	Ready to receive data
`recv_body_poll(exped)`	Check if data received

followed by one or more calls to `recv_header_poll()`. As soon as the header is completely received, the call-back function *handler* gets executed. Thus, the execution flow returns temporarily back to the upper layers.

The idea is that the handler has to inspect the header, to prepare for the receipt of the message body and to actually extract the data from the network. The second step typically leads to building an appropriate array of vectors. The last step is realized by making a call to `recv_body_post` (followed by subsequent calls to `recv_body_poll`) with the array as a parameter.

4 Implementation and Performance

The MADELEINE interface is designed to be portable, to provide efficient RPC-like communications and to minimize the overhead introduced over the underlying network protocol. We now illustrate these first two properties by describing the implementation and performance of MADELEINE on top of VIA, TCP. The overhead added by MADELEINE over the underlying protocol is small and constant. This point is detailed in [2].

4.1 High portability

Thanks to its modular structure, the MADELEINE communication library has been ported on a number of network protocols. In particular, it is available on top of low-level protocols such as VIA [5], BIP [14], SCI [6] and SBP [15]. It is also available on top of TCP/IP (for portability), PVM [8] and MPI [11][1]. This ability to adapt to very low-level protocols is the key to get the maximum performance of the underlying network. We now illustrate this point by presenting the implementation and performance of MADELEINE on top of VIA over a Fast-Ethernet network.

MADELEINE over the VIA Interface MADELEINE is one of the first high-level communication software available on top of VIA (Virtual Interface Architecture [5]), the interface proposed by Compaq, Intel and Microsoft for the management of high performance networks. The VI Architecture provides the user application with a protected, directly accessible interface to the network hardware: the *Virtual Interface* (VI). Once connected to another VI, a VI allows bi-directional point-to-point data transfer. Each VI contains a *Send Queue* and a *Receive Queue* where communication requests have to be posted. Such requests are described by *descriptors* that contain pointers to user data buffers. Data can be transmitted via regular message passing mechanisms or via remote direct memory access (read/write) operations. Before a descriptor is posted, all the data buffers it references should be explicitely *registered* in the VIA runtime.

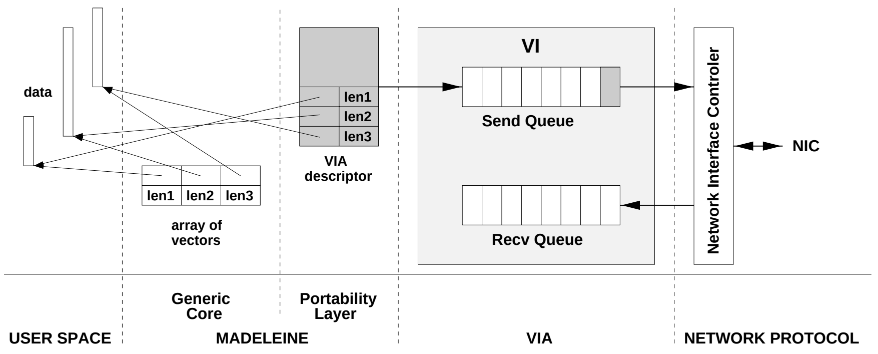

Fig. 6. Sending data in the VIA implementation of MADELEINE.

[1] Some vendors implementations of these libraries are very efficient (*e.g.* MPI-F on IBM SP2 or PVMe on Cray T3D).

The implementation of the portability layer on top of this VI architecture is easy and efficient, because gather/scatter functionalities are provided with the descriptor management. Figure 6 shows the typical execution path during the sending of a message *body* formed out of three vectors (see Section 3.2). The madeleine generic message management code first builds an array describing the message. This array is passed to the portability layer which is VIA specific in this case. A single VIA descriptor is then built by only copying the three entries of the array of vectors. At this point, the memory areas containing the user data are registered in VIA. Finally, the descriptor is posted in the Send Queue (which is an asynchronous operation) and the current thread can yield the processor until the operation is completed.

The implementation of a whole message sending is slightly more elaborate because VIA does not provide flow-control. Thus, before a descriptor like the previous one can be posted in a Send Queue, one has to make sure that a corresponding descriptor has already been posted in the destination Receive Queue. Madeleine uses acknowledgment messages to control the posting of message bodies. As far as headers are concerned, a simple credit-based algorithm is used so that acknowledgement messages can be avoided in many situations.

We have used the M-VIA implementation of VIA (developped at Berkeley Lab, `http://www.nersc.gov/research/FTG/via`) on a Fast-Ethernet (100Mb/s) network to evaluate the advantage of using a low-level high-performance protocol in comparison to using TCP/IP[2]. Figure 7 shows the performance (latency) of raw message passing over Madeleine using M-VIA and TCP as underlying protocols. That for, a ping-pong test was run between two Pentium133MHz PCs running Linux.

As one can see, the gap between the two curves is important, which confirms the interest of being portable on low-level protocols. We achieve a 37 μs latency with VIA (140 μs with TCP) and the bandwith for 64 KB messages is greater than 11.5 MB/s.

The PM^2 multithreaded environment, which used Madeleine for communications, directly benefits from this performance. The PM^2 thread migration time over VIA on this same architecture is 150 μs.

4.2 Efficiency of RPC operations

To illustrate the adequacy of Madeleine for RPC-based environments, we have measured the performance of processing remote procedure calls respectively over Madeleine and over MPI. The parameter of the procedure call is an array of bytes which is transmitted without any extra copy. This experiment has been done on a pile of PCs (Intel PentiumPro 200MHz with 64 MB of memory running Linux) interconnected by a Myrinet network.

As discussed previously (Section 2), the only way of realizing such a remote invocation with MPI is to send two messages. The first one carries the header (containing the procedure Id and the array size) and the second one transports

[2] The TCP implementation of Madeleine is described in the following section.

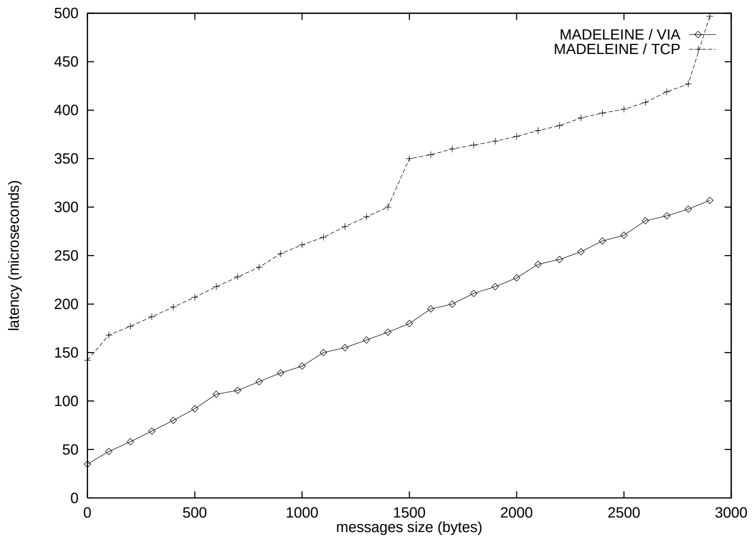

Fig. 7. Performance of MADELEINE on top of M-VIA and TCP over a 100Mb/s Ethernet network.

the array itself. With MADELEINE, we used the `RECV_EXPRESS` mode to pack the array size and the `SND_CHEAPER+RECV_CHEAPER` mode to pack the array itself (as in the example shown in Figure 4).

The next section compares the two approaches (MADELEINE vs. MPI) on top of the TCP/IP protocol. First, the TCP implementation of MADELEINE is described. Then, we report on some performance results.

MADELEINE over the TCP Protocol The implementation of MADELEINE over TCP, which provides flow control, is straightforward. The sending primitives of the portability layer (see Table 1) are just implemented by calls to the non-blocking `writev` Unix primitive. The message (header + body) is thus simply sent as a stream of bytes which is regulated by the internal TCP flow control algorithm. On the receiving side, the detection of the message availability (`recv_header_poll`) is realized through a call to the Unix `select` primitive. Then, the message header is extracted by means of a `read` operation. Finally, the message body can be extracted with a `readv` operation. As a result, no extra message nor extra copy is introduced by MADELEINE on top of TCP.

Figure 8 reports the times measured from the initialization of the communication on the source machine to the effective beginning of procedure execution on the destination machine. These times correspond to various array sizes as indicated on the abscissa.

As one can see, the gap in performance between MADELEINE and MPI is huge and proportional to message size (times obtained with MPI are approximately

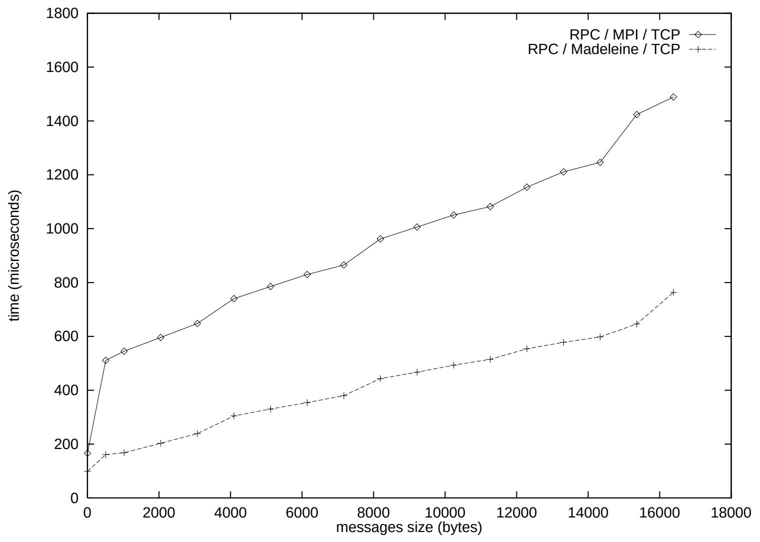

Fig. 8. Performance of Remote Procedure Calls over Madeleine and over MPI. The underlying communication protocol is TCP/IP.

twice the times obtained with Madeleine). This shows the better adequacy of the Madeleine interface to support RPC operations on buffering protocols. Here, the performance gap is due to the number of TCP messages exchanged with the MPI version. In fact, Madeleine needs only one TCP message to process the RPC, whereas the MPI interface requires the application to send two "user" messages. In addition, the internal flow-control algorithm of the MPI implementation may even trigger the sending of a third message from the receiver back to the sender. It is important to note that this gap is not in any way due to the "quality" of the MPI implementation. It simply reveals that the MPI interface lacks functionalities as far as RPC-like operations are concerned. Of course, experiments featuring other buffering protocols (such as SBP [15]) lead to similar results.

5 Related Work

Many communication libraries have been recently designed to provide portable interfaces and/or efficient implementations to build distributed applications. Among them, the FM (Illinois Fast Messages) [13] communication interface presents some similarities with the Madeleine portability layer. FM is derived from the AM (Active Messages) communication software designed at Berkeley [16]. It could be considered as a "medium-level" communication layer in that sense that it can be ported on top of previously mentioned low-level ones. Its interface provides a very simple mechanism to send data to a receiving node that is notified upon arrival by the activation of a handler. The 2.0 release of this interface provides some gather/scatter features with the introduction of a "streaming message" concept that allows the sending and the receiving of data in several (not necessarily equal) pieces. This feature may allow an efficient implementation of zero-copy RPC operations as presented in section 2. However, this interface still provides the user with a low abstraction level: data insertion/extraction operations are always synchronous and put a strong constraint on network management. We have seen that MADELEINE does not have this drawback while providing a higher level of abstraction to the upper layers.

6 Conclusion and future work

Most existing multithreaded environments providing RPC-based functionalities are implemented on message-passing communication interfaces such as PVM or MPI for portability reasons. However, we have shown these interfaces are not adequate for this purpose because they lacks expressiveness as far as RPC operations are concerned. Thus, even if their implementation is efficient on a given network technology, applications that need RPC facilities may not be able to achieve much of the underlying hardware's performance.

We have proposed a new portable communication interface which better meets the needs of these environments. MADELEINE is composed of a programming interface providing high-level functionalities to upper-level software, and a portability interface hiding the network specifics. We have explained how these interfaces allows efficient (zero-copy) data transmissions during RPC operations.

We have implemented MADELEINE on top of several low-level network interfaces and have reported its performance on top of VIA and TCP/IP. These experiments have demonstrated the superiority of MADELEINE over classical message-passing interfaces. They also have proved that MADELEINE can be easily implemented on low-level protocols while achieving much of the underlying hardware's performance. MADELEINE is currently available on top of the following network interfaces : VIA, BIP, SBP, Dolphin-SCI, TCP, PVM and MPI. The implementation of an existing multithreaded environment (PM^2) has been successfully ported on top of MADELEINE. The performance of PM^2 on top of MADELEINE validates our work: with a BIP-based implementation on a PentiumPro 200MHz cluster connected by Myrinet, a null RPC takes 11 μs and a thread migration takes 65 μs.

In the near future, we intend to extend the MADELEINE portability interface in order to optimally exploit network protocols providing fixed-size preallocated buffers, such as SBP [15]. We also plan to implement a generic flow control algorithm in the high-level layer of MADELEINE, so that new portability layers would become even easier to implement. Finally, we plan to extend the programming interface in order to provide a better support for SCI networks.

References

1. Boden, N., Cohen, D., Feldermann, R., Kulawik, A., Seitz, C., and Su, W. Myrinet: A Gigabit per second Local Area Network. *IEEE-Micro 15-1* (feb 1995), 29–36. Available from *http://www.myri.com/research/publications/Hot.ps*.
2. Bougé, L., Méhaut, J.-F., and Namyst, R. Madeleine: an efficient and portable communication interface for multithreaded environments. In *Proc. 1998 Int. Conf. Parallel Architectures and Compilation Techniques (PACT'98)* (ENST, Paris, France, Oct. 1998), IFIP WG 10.3 and IEEE, pp. 240–247.
3. Briat, J., Ginzburg, I., Pasin, M., and Plateau, B. Athapascan runtime : Efficiency for irregular problems. In *Proceedings of the Europar'97 Conference* (Passau, Germany, 1997), Springer Verlag, pp. 590–599.
4. Bubeck, T., and Rosenstiel, W. Verteiltes Rechnen mit DTS (Distributed Thread System). In *Proc. '94 SIPAR-Workshop on Parallel and Distributed Computing* (Suisse, Oct. 1994), M. Aguilar, Ed., Fribourg, pp. 65–68.
5. Compaq, Intel, and Microsoft. *Virtual Interface Architecture Specification*, December 1997. Version 1.0.
6. Dolphin Interconnect Solutions Inc. *PCI-SCI Adapter Programming Specification*, March 1997.
7. Foster, I., Kesselman, C., and Tuecke, S. The Nexus approach to integrating multithreading and communication. *Journal on Parallel and Distributed Computing*, 37 (1996), 70–82.
8. Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Mancheck, R., and Sunderam, V. *PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing*, 1994.
9. Haines, M., Cronk, D., and Mehrotra, P. On the design of chant: A talking threads package. In *Proc. of Supercomputing'94* (Washington, November 1994), pp. 350–359.
10. Lauria, M., and Chien, A. MPI-FM: High performance MPI on workstation clusters. *Jounal on Parallel and Distributed Computing*, 40 (01) (1997), 4–18.
11. Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, March 1994. available from www.mpi-forum.org.
12. Namyst, R., and Mehaut, J. PM^2: Parallel Multithreaded Machine. a computing environment for distributed architectures. In *ParCo'95 (PARallel COmputing)* (Sep 1995), Elsevier Science Publishers, pp. 279–285.
13. Pakin, S., Lauria, M., and Chien, A. High Performance Messaging on Workstations : Illinois Fast Messages (FM) for Myrinet. In *Proc. of Supercomputing'95* (San Diego, California, December 1995). Available from *http://www-csag.cs.uiuc.edu/papers/myrinet-fm-sc95.ps*.
14. Prylli, L., and Tourancheau, B. BIP: A New Protocol designed for High-Performance Networking on Myrinet. In *Proc. of PC-NOW IPPS-SPDP98* (Orlando, USA, March 1998).
15. Russell, R., and Hatcher, P. Efficient kernel support for reliable communication. In *13th ACM Symposium on Applied Computing* (Atlanta, GA, February 1998). To appear.
16. von Eicken, T., Culler, D., Goldstein, S., and Schauser, K. Active messages: a mechanism for integrated communication and computation. In *Proc. 19th Int'l Symposium on Computer Architecture* (May 1992).