A Portable and Adaptative Multi-Protocol Communication Library for Multithreaded Runtime Systems

Olivier Aumage, Luc Bougé, and Raymond Namyst

LIP, ENS Lyon, France*

Abstract. This paper introduces *Madeleine II*, an adaptive multi-protocol extension of the *Madeleine* portable communication interface. *Madeleine II* provides facilities to use multiple network protocols (VIA, SCI, TCP, MPI) and multiple network adapters (Ethernet, Myrinet, SCI) within the same application. Moreover, it can dynamically select the most appropriate transfer method for a given network protocol according to various parameters such as data size or responsiveness user requirements. We report performance results obtained using Fast-Ethernet and SCI.

1 Efficient Communication in Multithreaded Environments

Due to their ever-growing success in the development of distributed applications on clusters of SMP machines, today's multithreaded environments have to be highly portable and efficient on a large variety of architectures. For portability reasons, most of these environments are built on top of widespread message-passing communication interfaces such as PVM or MPI. However, the implementation of multithreaded environments mainly involves *RPC-like* interactions. This is obviously true for environments providing a RPC-based programming model such as Nexus [2] or PM2 [4], but also for others which often provide functionalities that can be efficiently implemented by RPC operations.

We have shown in [1] that message passing interfaces such as MPI, do not meet the needs of RPC-based multithreaded environments with respect to efficiency. Therefore, we have proposed a portable and efficient communication interface, called *Madeleine*, which was specifically designed to provide RPC-based multithreaded environments with *both* transparent and highly efficient communication. However, the internals of this first implementation were strongly message-passing oriented. Consequently, the support of non message-passing network protocols such as SCI or even VIA was cumbersome and introduced some unnecessary overhead. In addition, no provision was made to use multiple network protocols within the same application. For these reasons, we decided to design *Madeleine II*, a full multi-protocol version of *Madeleine*, efficiently portable on a wider range of network protocols, including non message-passing ones.

* LIP, ENS Lyon, 46, Allée d'Italie, F-69364 Lyon Cedex 07, France. Contact: `Raymond.Namyst@ens-lyon.fr`.

`mad_begin_packing`	Initiates a new message
`mad_begin_unpacking`	Initiates a message reception
`mad_end_packing`	Finalize an emission
`mad_end_unpacking`	Finalize a reception
`mad_pack`	Packs a data block
`mad_unpack`	Unpacks a data block

Table 1. Functional interface of *Madeleine II*.

2 The *Madeleine II* Multi-Protocol Communication Interface

The *Madeleine II* programming interface provides a small set of primitives to build RPC-like communication schemes. Theses primitives actually look like classical message-passing-oriented primitives. 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 (Table 1).

A message consists of several pieces of data, located anywhere in user-space. They are constructed (resp. de-constructed) incrementally using *packing* (resp. *unpacking*) primitives, possibly at multiple software levels, 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).

The critical point of a send operation is obviously the series of *packing* calls. Such packing operations simply *virtually* append the piece of data to a message under construction. In addition to the address of data and its size, the packing primitive features a pair of *flag* parameters which specifies the semantics of the operation. The available emission flags are the following:

`send_SAFER` This flag indicates that *Madeleine II* should pack the data in a way that further modifications to the corresponding memory area should not corrupt the message. This is particularly mandatory if the data location is reused before the message is actually sent.

`send_LATER` This flag indicates that *Madeleine II* should not consider accessing the value of the corresponding data until the `mad_end_packing` primitive is called. This means that any modification of these data between their packing and their sending shall actually update the message contents.

`send_CHEAPER` This is the default flag. It allows *Madeleine II* to do its best to handle the data as efficiently as possible. The counterpart is that no assumption should be made about the way *Madeleine II* will access the data. Thus, the corresponding data should be left unchanged until the send operation has completed. Note that most data transmissions involved in parallel applications can accommodate the `send_CHEAPER` semantics.

The following flags control the reception of user data packets:

`receive_EXPRESS` This flag forces *Madeleine II* to guarantee that the corresponding data are immediately available after the *unpacking* operation. Typically, this flag is mandatory if the data is needed to issue the following *unpacking* calls. On some network protocols, this functionality may be available for free. On some others, it may put a high penalty on latency and bandwidth. The user should therefore extract data this way only when necessary.

`receive_CHEAPER` This flag allows *Madeleine II* to possibly defer the extraction of the corresponding data until the execution of `mad_end_unpacking`. Thus, no assumption can be made about the exact moment at which the data will be extracted. Depending on the underlying network protocol, *Madeleine II* will do its best to minimize the overall message transmission time. If combined with `send_CHEAPER`, this flag guarantees that the corresponding data is transmitted as efficiently as possible.

Figure 1 illustrates the power of the *Madeleine* interface. Consider sending a message made of an array of bytes whose size is unpredictable on the receiving side. Thus, on the receiving side, one has first to extract the size of the array (an integer) before extracting the array itself, because the destination memory has to be dynamically allocated. In this example, the constraint is that the integer must be extracted `EXPRESS` *before* the corresponding array data is extracted. In contrast, the array data may safely be extracted `CHEAPER`, striving to avoid any copies.

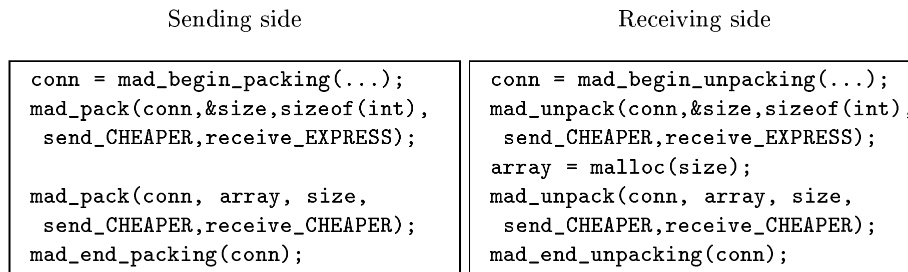

Fig. 1. Sending and receiving messages with *Madeleine II*.

Madeleine II aims at enabling an efficient and exhaustive use of underlying communication software and hardware functionalities. It is able to deal with several network protocols within the same session and to manage multiple network adapters (NIC) for each of these protocols. The user application can dynamically and explicitly switch from one protocol to another, according to its communication needs. The multi-protocol support of *Madeleine II* relies on the concept of *channel*.

Channels in *Madeleine II* are pretty much like radio channels. They are allocated at run-time. The communication on a given channel does not interfere with the communication on another one. As a counterpart, in-order delivery is not guaranteed among distinct channels. In-order delivery is only enforced for

```
text_chan  = mad_open_channel(TCP_ETH0);
video_chan = mad_open_channel(SISCI_SCI0);
text_conn  = mad_begin_packing(text_chan, video_client);
video_conn = mad_begin_packing(video_chan, video_client);
mad_pack(text_conn, text_dataptr, text_len, ...);
mad_pack(video_conn, video_dataptr, video_len, ...);
...
```

Fig. 2. Example of a video server simultaneously sending video information using a SISCI channel and translation text data using TCP channel.

point-to-point connections within the same channel. In this respect, they look like MPI communicators, but different *Madeleine II* channels can be bound to different protocols as well as adapters (Fig. 2). Of course, several channels may share the same protocol, and even the same adapter.

3 Inside *Madeleine II*: from the Application to the Network

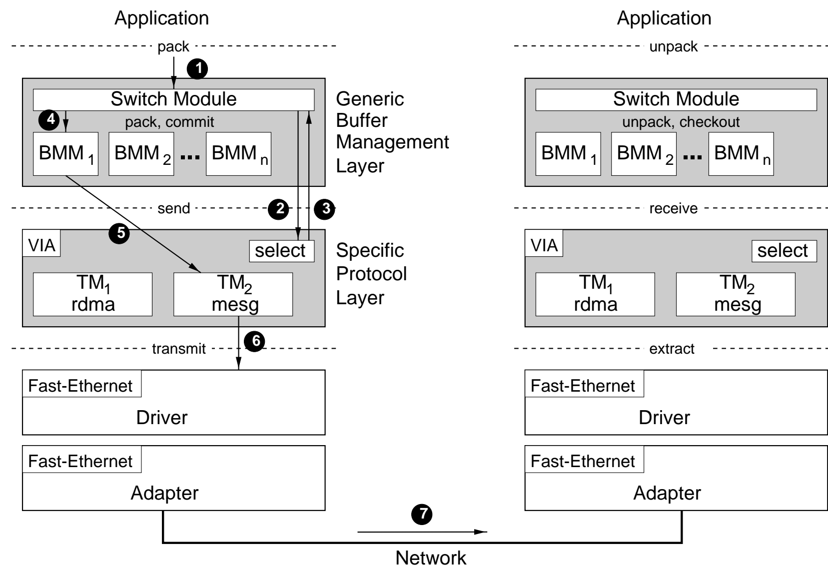

Fig. 3. Conceptual view of the data path through *Madeleine II*'s internal modules.

The transmission of data blocks using *Madeleine II* involves several internal modules. We illustrate its internals in the case of an implementation on top of VIA (Fig. 3).

Protocols such as VIA provide several methods to transfer data, namely regular message passing and remote DMA write (and optionally RDMA-read).

Moreover, there are several ways to use these transfer methods, as VIA requires registering the memory blocks before transmission. It is for instance possible to dynamically register user data blocks, or to copy them into a pool of pre-registered internal buffers. Their relative efficiency crucially depends on the size of the blocks. The current implementation of *Madeleine II* on top of VIA supports the three following combinations:

Small blocks: message-passing + static buffer pool.
Medium-sized blocks: message-passing + dynamically registered buffers.
Large blocks: RDMA-write + dynamically registered buffers.

Each transfer method is encapsulated in a protocol-specific *Transmission Module* (TM, see Fig. 3). Each TM is associated with a *Buffer Management Module* (BMM). A BMM implements a generic, protocol-independent management policy: either the user-allocated data block is directly referenced as a buffer, or it is copied into a buffer provided by the TM. Moreover, each BMM implements a specific scheme to aggregate successive buffers into a single piece of message. Each TM is associated with its optimal BMM. However, observe that several TM (even from different protocols) may share the same BMM, which results in a significant improvement in development time and reliability.

In the case of VIA, one can for instance take advantage of the gather/scatter capabilities of VIA to issue one-step burst data transfers when possible. This strategy is rewarding for *medium-size blocks* scattered in user-space. For *small blocks* accumulated into static buffers, it is most efficient to immediately transfer buffers as soon as they get full: this enhances pipelining and overlaps the additional copy involved.

Sending Side One initiates the construction of an outgoing message with a call to `begin_packing(channel, remote)`. The `channel` object selects the protocol module (VIA in our case), and the adapter to use for sending the message. The `remote` parameter specifies the destination node. The `begin_packing` function returns a `connection` object.

Using this `connection` object, the application can start packing user data into packets by calling `pack(connection, ptr, len, s_mode, r_mode)`. Entering the Generic Buffer Management Layer, the packet is examined by the *Switch Module* (Step 1 on Fig. 3). It queries the Specific Protocol Layer (Step 2) for the best suited *Transmission Module*, given the length and the send/receive mode combination. The selected TM (Step 3) determines the optimal *Buffer Management Module* to use (Step 4). Finally, the Switch Module forwards the packet to the selected BMM. Depending on the BMM, the packet may be handled as is (and considered as a buffer), or copied into a new buffer, possibly provided by the TM. Depending on its aggregation scheme, the BMM either immediately sends the buffer to the TM or delays this operation for a later time. The buffer is eventually sent to the TM (Step 5). The TM immediately processes it and transmits it to the Driver (Steps 6). The buffer is then eventually shipped to the Adapter (Step 7).

Special attention must be paid to guarantee the delivery order in presence of multiple TM. Each time the Switch Step selects a TM differing from the

previous one, the corresponding previous BMM is flushed (*commit* on Fig. 3) to ensure that any delayed packet has been sent to the network. A general *commit* operation is also performed by the `end_packing(connection)` call to ensure that no delayed packet remains waiting in the BMM.

Receiving Side Processing an incoming message on the destination side is just symmetric. A message reception is initiated by a call to `begin_unpacking(channel)` which starts the extraction of the first incoming message for the specified channel. This function returns the `connection` object corresponding to the established point-to-point connection, which contains the remote node identification among other things.

Using this `connection` object, the application issues a sequence of `unpack(connection, ptr, len, s_mode, r_mode)` calls, symmetrically to the series of `pack` calls that generated the message. The Switch Step is performed on each `unpack` and must select the same sequence of TM as on the sending side. For instance, a packet sent by the DMA Transmission Module of VIA must be received by the same module on the receiving side. The *checkout* function (dual to the *commit* one on the sending side) is used to actually extract data from the network to the user application space: indeed, just like packet sending could be delayed on the sending side for aggregation, the actual packet extraction from the network may also be delayed to allow for burst data reception. Of course, the final call to `end_unpacking(connection)` ensures that all expected packets are made available to the user application.

Discussion This modular architecture combined to packet-based message construction allows *Madeleine II* to be efficient on top of message-passing protocols as well as put/get protocols. Whatever the underlying protocol used, *Madeleine II*'s generic flexible buffer management layer is able to tightly adapt itself to its particularities, and hence deliver most of the available networking potential to the user application. Moreover, the task of implementing a new protocol into *Madeleine II* is considerably alleviated by re-using existing BMM.

4 Implementation and Performances

We now evaluate *Madeleine II* on top of several network protocols. All features mentioned above have been implemented. Drivers are currently available for TCP, MPI, VIA, SISCI [3] and SBP [6] network interfaces.

Testing Environment The following performance results are obtained using a cluster of dual Intel Pentium II 450 MHz PC nodes with 128 MB of RAM running Linux (Kernel 2.1.130 for VIA, and Kernel 2.2.10 for TCP and SISCI). The cluster interconnection networks are 100 Mbit/s Fast Ethernet for TCP and VIA, and Dolphin SCI for SISCI. The tests run on the TCP/IP protocol use the standard UNIX sockets. The tests run on the VIA protocol use the M-VIA 0.9.2 implementation from the NERSC (National Energy Research Scientific Computing Center, Lawrence Berkeley Natl Labs).

	Latency		Bandwidth	
Protocol	TCP	SISCI	TCP	SISCI
Raw performance	59.8 μs	2.3 μs	11.1 MB/s	76.5 MB/s
Madeleine	77.4 μs	5.9 μs	10.5 MB/s	70.0 MB/s
Madeleine II	67.2 μs	7.9 μs	11.0 MB/s	57.0 MB/s

Table 2. Latency (left) and bandwidth (right) on top of TCP and SISCI.

TCP Surprisingly enough, *Madeleine II* outperforms *Madeleine* (Table 2). *Madeleine* used to require attaching a short header to each transfered message, whereas *Madeleine II* gives the user finer control on the message structure. The difference in performance between raw TCP and *Madeleine II* on top of TCP is the result of the current software overhead of *Madeleine II*. The bandwidth of *Madeleine II* on top of TCP is very close to the raw bandwidth of TCP.

SISCI The new SISCI Specific Protocol Layer of *Madeleine II* is not yet as optimized as the one used by *Madeleine*. This is why the bandwidth measured with *Madeleine II* on top of SISCI is not as good as the one obtained with *Madeleine* (Table 2). The difference in latency between *Madeleine II* and *Madeleine* is due to some additional processing in the internals of *Madeleine II*. Future optimizations will hopefully solve this problem.

Dynamic Transfer Method Selection We mentioned above the capability of *Madeleine II* to dynamically choose the most appropriate transfer paradigm within a given protocol. Figure 4 shows the dramatic influence of dynamic transfer paradigm selection on performance using VIA. VIA requires the memory areas involved in transfer to be registered. Such dynamic registration operations are expensive. This cost is especially prohibitive for short messages, and using a pool of pre-registered buffers help circumventing the problem. Instead of registering the memory area where the messages are stored, one can copy the messages into these buffers. This amounts to exchanging registration time for copying time. This is obviously inefficient for long messages. The two curves are plotted on Figure 4. The *Multi-Paradigm* curve is obtained by activating the dynamic paradigm selection of *Madeleine II*. It is optimal *both* with short messages and long messages!

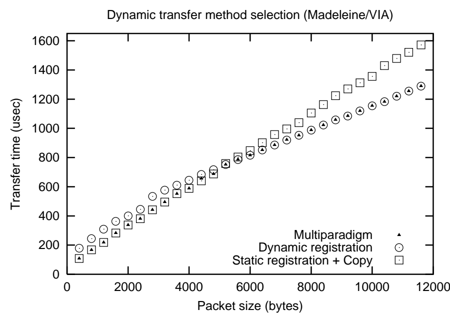

Fig. 4. Multi-Paradigm support.

5 Related work

Many communication libraries have recently been designed to provide portable interfaces and/or efficient implementations to build distributed applications.

However, very few of them provide an efficient support for RPC-like communication schemes, support for multi-protocol communications and support for multithreading.

Illinois Fast Messages (FM) [5] provides a very simple mechanism to send data to a receiving node that is notified upon arrival by the activation of a handler. Releases 2.x of this interface provide interesting gather/scatter features which allow an efficient implementation of zero-copy data transmissions. However, it is not possible to issue a transmission with the semantics of the `receive_CHEAPER` *Madeleine II* flag: only `receive_EXPRESS`-like receptions are supported, and it is not possible to enforce aggregated transmissions.

The Nexus multithreaded runtime [2] features a multi-protocol communication subsystem very close to the one of *Madeleine II*. The messages are constructed using similar *packing* operations except that no "high level" semantics can be associated to data: there is no notion of `CHEAPER` specifications, which allows *Madeleine II* to choose the best suited strategy. Also, as for FM, *unpacking* operations behave like `receive_EXPRESS` *Madeleine II* transmissions.

6 Conclusion

In this paper, we have described the new *Madeleine II* communication interface. This new version features full multi-protocol, multi-adapter support as well as an integrated new dynamic *most-efficient transfer-method* selection mechanism. We showed that this mechanism gives excellent results with protocols such as VIA. We are now actively working on having *Madeleine II* running across clusters connected by heterogeneous networks.

References

1. Luc Bougé, Jean-François Méhaut, and Raymond Namyst. Efficient communications in multithreaded runtime systems. In *Proc. 3rd Workshop on Runtime Systems for Parallel Programming (RTSPP '99)*, volume 1586 of *Lect. Notes Comp. Science*, pages 468–182, San Juan, Puerto Rico, April 1999. Springer-Verlag.
2. I. Foster, C. Kesselman, and S. Tuecke. The Nexus approach to integrating multithreading and communication. *Journal on Parallel and Distributed Computing*, 37(1):70–82, 1996.
3. IEEE. *Standard for Scalable Coherent Interface (SCI)*, August 1993. Standard no. 1596.
4. Raymond Namyst and Jean-François Méhaut. PM2: Parallel Multithreaded Machine. a computing environment for distributed architectures. In *Parallel Computing (ParCo'95)*, pages 279–285. Elsevier, September 1995.
5. S. Pakin, V. Karamcheti, and A. Chien. Fast Messages: Efficient, portable communication for workstation clusters and MPPs. *IEEE Concurrency*, 5(2):60–73, April 1997.
6. R.D. Russell and P.J. Hatcher. Efficient kernel support for reliable communication. In *13th ACM Symposium on Applied Computing*, pages 541–550, Atlanta, GA, February 1998.