

MMPS: Portable Message Passing Support for Parallel Computing

Andrew Grimshaw, David A. Mack, W. Timothy Strayer

Department of Computer Science
University of Virginia, Charlottesville, VA

Abstract

Models of parallel computation based upon message passing are in wide-spread use today, yet the message passing primitives available on different architectures are often different in subtle ways. The situation on distributed systems is even worse; not only are there different interfaces, but the services provided are not sufficient for data driven computation. MMPS is a solution to the problem. First, MMPS provides a basic message passing service with guaranteed delivery that can be ported to a wide variety of architectures. Applications that use the MMPS interface will be portable with respect to the message system. Second, MMPS provides a customizable interface that exploits the C++ [1] class hierarchy to allow the user to define new types of messages with new services. The new services can be efficiently implemented using existing code and inheritance.

1. Introduction

There are many different parallel processing architectures that use the message passing paradigm on the market today. Applications writers are faced with rewriting that portion of the application that interfaces with the message passing library each time they port their application from one architecture to another. Subtle, and not so subtle, differences in the message system's syntax and semantics may not be fully understood, leading to unpredictable errors and frustrating experiences. Furthermore, they must often customize the message passing services provided to match application requirements. This is an annoying and unnecessary waste of programmer resources.

The problem of porting between architectures is even worse when the target architecture's system software was not designed with parallel processing in mind. This is the case in many distributed systems. Not only are basic reliable messages services not available [2,3], but when they are, they are often based on a call/return semantics [4,5,6]. For example, the services provided to Berkeley UNIX users are unreliable datagrams of fixed size (UDP), and totally reliable streams of

characters (TCP). Both of these are layered on top of the IP. This has been adequate in the past when the primary model of distributed computation was one of using either pipes between machines (or virtual terminal sessions) or using RPC in which a time-out could be used. These services are totally inadequate for writing highly-parallel, asynchronous applications.

What is needed is a portable, customizable, message passing system that provides message passing primitives designed to support parallel computation. If such a message passing system were available on several different architectures, then applications could be ported between the supported architectures with little or no modification. Besides the savings in porting effort, it would also be possible to compare the architectural performance using the same application code.

The Modular Message Passing System (MMPS) is just such a system. The MMPS solution has two important aspects: a set of primitive message passing routines that can be readily ported from architecture to architecture, and an extensible set of C++ classes from which different message classes with different service characteristics can be easily and efficiently constructed using inheritance.

In the next section we introduce and describe the extensibility/customization features of MMPS. This includes a description of the C++ inheritance mechanism used, the class definitions used, and two examples illustrating the flexibility of the approach. Next we describe our portability philosophy and the two current instances of MMPS, on a network of UNIX BSD hosts, and on the Intel iPSC/2 hypercube. Finally, we offer performance results for MMPS in the two supported environments, and compare them to the native communication primitives.

2. Extensibility Features

It is important to provide an extensible message passing system since there are such diverse requirements for message passing services among applications. Rather than trying to provide a

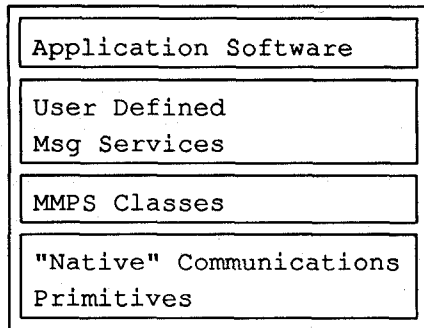


Figure 1 — Layers.

union of all such services, MMPS provides a basic message service via a predefined C++ class, the *mmps_message* class. The use of the C++ class inheritance mechanism allows users to customize the message system to provide the services needed without building an entire application-dependent service from scratch. The basic services are provided by the *mmps_message* class. The user can then derive new message classes from *mmps_message* that provide the specific services that he desires in much the same manner as traditional message systems use layers of protocols. For example, the user may want to provide a software multicast as a new *send* service, *send(name_list)*. The advantage of using the C++ class inheritance mechanism is that the "layering" of protocols is very efficient, and can be done at compile time. This "layering" is shown in Figure 1. The upper three layers are compiled layers; interlayer communication is done via procedure calls. The lowest layer is the native layer; interlayer communication is done via IPC or traps.

2.1. Inheritance in C++

Classes in C++ are declared with the keyword *class*. A class definition consists of a *public* section and a *private* section. Each section may contain instance variables, called *member variables* in C++, and functions defined for the class, called *member functions*. The public members define the interface to the class. Each class may have a *constructor* and a *destructor* declared. The constructor and destructor are automatically invoked when a new instance of the class comes

into existence, and when an instance is destroyed, respectively.

A class DERIVED inherits from its base class BASE all of the public and private member variables and functions of the class BASE. This is shown syntactically in Figure 2. The in-memory representation of DERIVED is as if the two declarations (BASE and DERIVED) were concatenated in contiguous memory. We exploit this fact in our implementation. Instances of classes are "constructed" from the bottom up, first the base class constructor is called, then the derived.

An advantage of the C++ implementation of inheritance is that it allows the compile time binding of member function invocations. (The only exception is for virtual functions, for which a single indirect call is required.) The binding of procedures at compile time means that the overhead of using multiple "layers" via the inheritance mechanism is minimal.

2.2. MMPS Class and Inheritance

The *mmps_message* class interface provides services that are basic to parallel computation, namely, the reliable delivery of messages of arbitrary size from one nameable entity to another. The user may specify that each message contain different amounts of user specified header in addition to the basic message contents. This additional header space can be used by derived classes to store their class-specific information. Figure 3

```

class BASE {
    int foo,y,z;
public:
    int bar();
    BASE();    // constructor
    ~BASE();   // destructor
};
class DERIVED: public BASE {
    int w,x;
public:
    int setx(int arg);
    DERIVED(); // constructor
};
  
```

Figure 2 — Example of class inheritance in C++.

```

class mmmps_name {
public:
    char host_name[30];
    int RESOLVED_FLAG;
    int port; int pid;
};
class mmmps_message {
    mess_header b_header;
    int type;
    int valid;
    long size;
    int header_size;
    mmmps_name destination;
    mmmps_name source;
public:
    mmmps_message(long,int);
    ~mmmps_message();
    void set_type(int);
    int get_type();
    void get_data(long,char*);
    char* get_data_ptr();
    long get_data_length();
    long get_header_length();
    void put_data(long,char*);
    void set_dest(mmmps_name);
    mmmps_name *get_dest();
    mmmps_name *get_src();
    void send();
};
mmmps_message
    *mmmps_receive(int block);

```

Figure 3. The *mmmps_message* class definition.

shows the *mmmps_message* class interface. Each of the member functions is briefly described below.

- The constructor *mmmps_message()* takes as parameters the size of the data portion of the message and the size or derived class header. The constructor can only be called dynamically, so no auto variables are allowed. The constructor will create a *mmmps_message* object with space allocated for data and derived class header.
- The destructor *~mmmps_message()* releases the message back to message system. Once the user is finished with a message it should be released.

- The *set_type()* call is used to set the type of the message.
- The *get_type()* call returns the type of the message so that casts and assignments can be done correctly.
- The *get_data()* call takes as parameters a pointer to a char buffer and the length of that buffer. It will copy the data up to the given length into the buffer provided.
- The *get_data_ptr()* call returns the address of the data portion of the message.
- The *get_data_length()* call returns the length of the data portion of the message.
- The *get_header_length()* call returns the length of the derived class header.
- The *put_data()* call takes as parameters a pointer to a character buffer and the size of that buffer. It will copy up to the size bytes into the buffer.
- The *set_dest()* call sets the destination address for the message.
- The *get_dest()* call returns the destination of the message.
- The *send()* call sends the message. Checks are made to be sure that the destination is set and the message has data.
- The *mmmps_receive()* call receives a *mmmps_message* and returns a pointer to it. A parameter is used to indicate whether the receive is blocking or non-blocking.

The user may use the *mmmps_message* class to derive new classes that provide both additional services as well as overloaded service definitions. When this is done the memory representation of

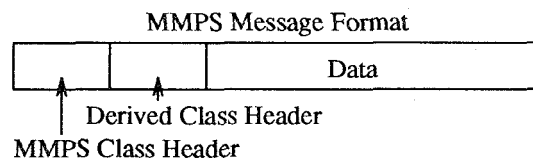


Figure 4 — The *mmmps_message* class inheritance.

```

class mentat_message:
    public mmmps_message {
        arc_instance destination;
        int f_size, f_space;
        fl_rec s_future[1];
    public:
        mentat_message(long, int);
        void set_dest(arc_instance);
        arc_instance *get_dest();
        mentat_message
            *set_future(fl_rec*,int);
        fl_rec *get_future();
        int future_size();
        void send();
};

mentat_message::
    mentat_message(long max_size,
        int fl_size=SHORT_FUTURE):
/* constructor parms first */
    (max_size,
        ((sizeof(mentat_message)
        -sizeof(mmmps_message))
        +(((fl_size>0?fl_size:1)-1)
        *sizeof(fl_rec))))
{ /* prevent memory allocation */
    if (this == 0) this = 0;
    f_size=(fl_size>0?fl_size:0);
    f_space=f_size;
}

```

Figure 5 — Mentat message class and constructor.

```

class mcast_message:
    public mmmps_message {
    public:
        void send(name_list *dests);
};

void mcast_message::
    send(name_list *dests) {
    int i;
    for (i=0;i<dests.count();i++) {
        set_dest(dests[i]);
        send();
    }
}

```

Figure 6 — Multicast message class.

the derived class is as shown in Figure 4.

2.3. Inheritance Examples

Our first example comes from the Mentat run-time system [7]. In addition to the basic services the Mentat run-time system requires messages that have additional variable-length data structures attached to every message. These structures specify (1) that a particular argument to a particular function is the destination of the message, and (2) a program graph that specifies where the receiver should send its reply. This information is contained in the `arc_instance` destination and `fl_rec s_future[]`, respectively. In addition to the data structures there are routines to manipulate them, and an overloaded `send()` with slightly different semantics (it deletes the message after it has been sent). The class definition for `mentat_message` is given in Figure 5, as is one of the constructors. Note how the constructor specifies to the base class how much header space is required.

The second example (shown in Figure 6) illustrates how a simple, reliable multicast message system could be constructed. The `name_list` class is a list of names that can be addressed by the overloaded array index operator.

3. Portability

The MMPS philosophy of portability is to keep most of the architectural dependencies isolated in the lowest layer that provides the basic transport service. This philosophy is not unique to MMPS; most modern operating systems strive to minimize and isolate machine dependencies. Thus, when MMPS is ported to a new architecture, only the lowest layer need be modified, the rest of the system, including all of the user defined message classes, port without modification due to the use of the class inheritance mechanism. (This is not true if a derived class includes architecturally specific information in its implementation.)

The only difference in MMPS that the user sees between architectures is in *naming*. What constitutes a nameable entity varies from system to system. For example, an internet name is a `hostname/port` pair, while on the Intel hypercube a name is a `node/pid` pair. Our solution to the problem is to encapsulate names into a `mmmps_name` class. This class is chosen at compile time by using `#ifdef` compiler directives. In addition to the structural changes, a set of

routines to manipulate the names is provided for each architecture. In the future, we will support a naming scheme that will permit exchanging messages between different "instances" of MMPS.

MMPS is currently available for two different systems: the Intel hypercube [8], and a network of SUN's running UNIX BSD. Both implementations provide the same services, although the Intel implementation is much faster. The SUN version is currently implemented using BSD sockets using UDP/IP [2,3]. In the future we plan to port MMPS to other architectures, including the BBN Butterfly (essentially a *Mach* port) and the NCUBE/ten.

3.1. Berkeley BSD

The BSD version of MMPS uses UDP/IP sockets and a protocol that establishes "connectionless" connections. We chose not to use a raw Ethernet system for our BSD port because it would be too architecture-specific, and hence would not be portable across the range of architectures we want to support. The connections are connectionless in that establishing a connection does not prevent the user from opening any new connections. Furthermore, the user can still communicate with all peers simultaneously. The protocol uses these connections to provide the reliability needed for MMPS.

There presently exist two network protocols available in Berkeley UNIX, TCP and UDP, both of which are built on the internet protocol, IP. TCP provides reliable service, but with a high overhead paid to establish connections and to perform error checking. UDP does not require the connection overhead of TCP, but does not provide the quality of service that TCP provides. In particular, when using UDP, packets have a fixed maximum length, may be lost, may arrive out of sequence, or may be duplicated. MMPS was designed to provide a service between TCP and UDP. MMPS is not a network protocol, nor is it ISO compliant, rather it is a message passing protocol that provides connectionless delivery of messages. It is built on top of UDP/IP sockets so that the high overhead of TCP need not be paid. Messages are delivered in-sequence and without duplicates.

To ensure delivery time-outs are needed, with timeouts comes the problem of duplicates. To deal with these problems MMPS establishes its own connection using datagrams and a stop-wait protocol. The connection lasts the duration

of the message and then broken-down again. The stop-wait protocol provides the guarantee of in-sequence delivery, and the connection provides duplicate suppression which allows timeouts to be used to guarantee delivery of messages.

3.2. Intel iPSC/2

The Intel iPSC/2 hypercube version of MMPS was much easier to implement than the Sun version. The iPSC/2 already provides message passing primitives that support parallel computation. The *mmps_message* class was "wrapped around" the iPSC/2 primitives to provide the extensibility and portability that the primitives lack.

4. Performance

MMPS provides communication support for parallel computation which is very comparable to existing communication services while providing more functionality and flexibility. Performance measurements of end-to-end data transfer delay for various size messages are shown for the Sun network of UNIX BSD hosts and for an Intel iPSC/2 hypercube. MMPS provides reliable services which are efficient and easily extensible via C++ inheritance, yet impose little additional cost.

The first graph in Figure 7 shows end-to-end delay measurements for MMPS data transfer services compared with TCP/IP and UDP/IP. Parallel applications on a distributed system like the network of Suns must currently rely on data transfer services provided by either TCP/IP or UDP/IP. For about one third of the cost in delay of TCP/IP, MMPS provides similar functionality while also providing more flexibility. For a constant overhead cost of about 8 milliseconds MMPS provides a reliable datagram service where UDP/IP is unreliable.

The second graph in Figure 7 shows end-to-end delay measurements for MMPS compared with the iPSC/2 hypercube. While MMPS data transfer services do not add value like reliability to the raw hypercube services, it only imposes a small overhead in exchange for the portability and extensibility it provides. Parallel applications can be easily ported between these two testbeds using MMPS; converting TCP/IP based service calls to hypercube service calls is far from trivial.

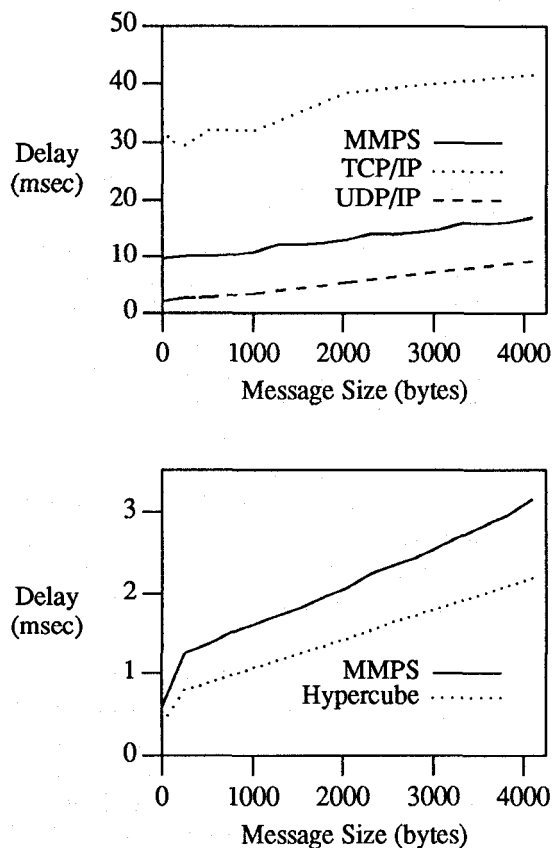


Figure 7 — MMPS end-to-end delay. Above: Network of Sun 3/60s. Below: Intel iPSC/2.

5. Conclusions

MMPS is a portable and easily customized message passing system that provides basic message passing services, including the reliable delivery of arbitrarily large messages. Systems such as MMPS are important to the distributed memory computing systems community since portability of applications between architectures will continue to require a major effort on the part of applications programmers. The vast difference between data transfer services within parallel machines and distributed systems further exacerbates this problem.

The extensibility for MMPS allows the user to customize a set of basic services at low cost for

the value added. This makes MMPS especially flexible for use in diverse architectures. Performance measurements of end-to-end delay show that MMPS data transfer services are very comparable to existing basic services. Extensible and portable services such as MMPS are required before significant progress can be made in architecture-independent applications.

References

- [1] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.
- [2] Sun Microsystems Inc., *SunOS Users Manual*, Mountain View, CA, 1988.
- [3] S. J. Leffer, R. S. Fabry, and W. N. Joy, "4.2BSD Interprocess Communication Primer," Computer Systems Research Group, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, July 1983.
- [4] D. R. Cheriton, "The V Kernel: A Software Base For Distributed Systems," *IEEE Software*, 1 (Apr.), 1984.
- [5] R. van Renesse, H. van Staveren, and A. Tannenbaum, "Performance of the World's Fastest Distributed Operating System," *ACM OSR*, vol 22, no. 4 (Oct.), 1984.
- [6] M. B. Jones, and R. Rashid, "Mach and Matchmaker: Kernel and Language Support for Object-Based Distributed Systems," Carnegie Mellon University, CMU-CS-87-150, September, 1987.
- [7] Grimshaw, A. S., "The Mentat Run-Time System: Support for Medium Grain Parallel Computation," *Proceedings of the Fifth Distributed Memory Computing Conference*, Charleston, SC., April 9-12, 1990.
- [8] Intel Corporation, "iPSC/2 USER'S GUIDE," Intel Scientific Computers, Beaverton, OR, March 1988.