MALLBA: Basic Module for Communications

Alba E., Cotta C., Díaz M., Soler E., Troya J.M.

January 30, 2001

Departamento de Lenguajes y Ciencias de la Computación
Universidad de Málaga
{eat, ccottap, mdr, esc, troya}@lcc.uma.es

Abstract

This paper contains a preliminar study on the most appropriate communication middleware to be used in the MALLBA project. Since our goal is to deliver a user friendly and geographically distributed optimization system we must analyze several issues on the best ways of building such a middleware. First, we review the potential advantages and drawbacks of some existing communication models and tools. Then, we discuss on the basic services required by our optimization system and give a possible service definition for the middleware that we call `NetStream`. This is a first stage in the development. A second phase could address more sophisticated services such as load balancing. Finally, the integration of the proposed middleware and the optimization skeleton being devised is studied with the aim of providing an integral solution to distributed optimization systems.

1 Introduction

This paper discusses the architecture and functionalities of a communication system that we call "middleware" to be used as the basic means for parallelizing and controlling the kind of processes needed in the project MALLBA (TIC1999-0754-C03-03).

Briefly stated, the MALLBA project is intended to provide a geographically distributed optimization system that allows a novice (though technical) user to pose a problem to be solved in parallel by many clusters of heterogeneous computers. Such a system could be build in many forms, but in MALLBA the optimization engines are embedded in the so-called "skeletons". A skeleton is a generic tool allowing the user to define a concrete optimization algorithm by creating instances of a general optimization procedure. The user is guided by a graphical interface to fill up the "holes" representing the alternatives the optimization algorithm provides to the user.

Skeletons for heuristic, exact, and hybrid algorithms will be developed within MALLBA, allowing a user to specify a problem-dependent algorithm that fits

his/her needs. Since the problems to which these skeletons will be faced can be quite difficult, parallel skeletons can also be specified, as extensions of the sequential ones. The parallel skeletons can run both on a NOW (Network Of Workstations) or in a geographically distributed system composed by several clusters of computers having different hardware capabilities.

After this brief revision of the MALLBA goals one point is clear: the design of the communication facilities for such a system is a very important issue. The communication services should be defined to be abstract enough for use in the skeletons and, at the same time, specific enough to the NOWs' technologies in order to get efficiency. By using the middleware services the skeleton designer will be able to spawn, trace, modify, and shut down the whole optimization task. This is the basic communication layer to be extended in order to build a more sophisticated service permitting automatic process location and taking into account the actual state of the whole hardware system (workstations, LAN and WAN links).

In short, we first need to review the existing communication models and toolkits in order to find out whether and how they match our requirements. Section 2 contains a brief discussion on this matter. Afterward, in Section 3, we present a first draft on the services of the middleware system. In Section 4 the integration between the optimization skeletons and the middleware library is analyzed. Finally, some conclusions and further work is pointed out in Section 5.

2 Evaluation of Existing Communication Systems

The principle objection to parallel computers and applications is that they are difficult to program. There is a significant component of truth in this claim, particularly for large-scale parallel machines. However, most scientific calculations require parallel computers to yield useful results in affordable run times.

There are three reasons why parallel programming is more challenging than sequential programming [19]. First, parallel programs must include the mechanics of exchanging data between processors or handling mutual exclusion regions. Second, in an efficient parallel program the work must be evenly divided among processors. Third, the data structures must be divided among processors to preserve data locality. The first reason adds complexity to the semantics and the syntax of a program, the second one is an algorithmic challenge with no serial counterpart, and the latter one is an extension of serial data locality and cache performance issues.

In this section we review some programming tools that allow parallel programming at different levels of flexibility and generality. We then discuss in a final subsection the advantages and drawbacks of the outlined systems from the point of view of our geographically distributed system. Obviously, there is a large number of such communication facilities, from which only a representa-

tive subset is included here. Now, let us proceed with an overview of popular communication models and tools.

2.1 Sockets

The BSD socket interface (see e.g. [3]) is a widely available message-passing programming tool. A set of data structures and C functions allow the programmer to establish full-duplex connections between two computers with TCP for implementing general purpose distributed applications. Also, a connectionless service over UDP (and even over IP) is available for applications needing such a facility. Synchronous and asynchronous parallel programs can be developed with the socket API, with the added benefits of large applicability, high standardization, and complete control on the communication primitives.

In despite their advantages, programming with sockets has many drawbacks for applications involving a large number of computers with different operating systems and on different networks. First, programming with sockets is error-prone and requires understanding low level characteristics of the network. Also, it does not include any process management, fault tolerance, task migration, security options, and other attributed usually requested in modern parallel applications.

In short, it is a great tool but its applications on modern internet and distributed systems requires a considerable effort to meet a satisfactory degree of abstraction.

2.2 PVM

The Parallel Virtual Machine (PVM) [14] is a software system that permits the utilization of a heterogeneous network of parallel and serial computers as a unified general and flexible concurrent computational resource. The PVM system supports the message passing paradigm, with implementations for distributed memory, shared-memory, and hybrid computers; thus, allowing applications to use the most appropriate computing model for the entire application of for individual sub-algorithms. Processing elements in PVM may be scalar machines, distributed and shared-memory multiprocessors, vector computers and special purpose graphic engines; thereby, permitting the use of the best-suited computing resource for each component of an application.

The PVM system is composed of a suite of user interface primitives supporting software that together enable concurrent computing on loosely coupled networks of processing elements. PVM may be implemented on heterogeneous architectures and networks. These computing elements are accessed by applications via a standard interface that supports common concurrent processing paradigms in the form of well-defined primitives that are embedded in procedural host languages. Application programs are composed of *components* that are sub-tasks at a moderately large level of granularity. During execution, multiple instances of each component may be initiated.

The advantages of PVM are its wide acceptability, and its heterogeneous computing facilities, including fault tolerance issues, and interoperability [15]. Managing a dynamic collection of potentially heterogeneous computational resources as a single parallel computer is the real appealing treat of PVM. In despite a large number of advantages, the standard for PVM has recently beginning to be unsupported (no further releases); also, users of the message-passing paradigm are shifting from using PVM to new models of such paradigm that run more efficiently on the new kinds of networks appearing nowadays. In addition, since PVM 3.4 (the latest version) does not support threads, applications targeted to shared-memory computers do not use this software.

2.3 MPI

The Message Passing Interface (MPI) is a library of message-passing routines [13]. When MPI is used, the processes in a distributed program are written in a sequential language such as C or Fortran; they communicate and synchronize by calling functions in the MPI library.

The MPI application programmer's interface (API) was defined in the mid-1990s by a large group of people from academia, government, and industry. The interface reflects people's experiences with earlier message-passing libraries, such as PVM. The goal of the group was to develop a single library that could be implemented efficiently on the variety of multiple processor machines. MPI has now become the *de facto* standard, and several implementations exist, such as MPICH `www.mcs.anl.gov/mpi/mpich` and LAM/MPI `www.mpi.nd.edu/lam`.

MPI programs have what is called an SPMD style - single program, multiple data. In particular, every processor executes a copy of the same program. Each instance of the program can determine its own identity and hence take different actions. The instances interact by calling MPI library functions. The MPI functions support process-to-process communication, group communication, setting up and managing communication groups, and interacting with the environment.

The MPI standard allows programmers to write message-passing programs without concern for low-level details such as machine type, network structure, low-level protocols, etc. MPICH provides a portable, high-performance implementation of MPI that incorporates some support for heterogeneous environments (e.g. the `p4` device), but provides only limited support for wide-area metacomputing environments [7]. Some extensions by using the Nexus communication library are available to interface with the Globus metacomputing toolkit, an ongoing research project with important implications for the parallel computing community (see the next section). This means that WAN facilities will be provided for MPI users.

The impetus for developing MPI was that each massively parallel processor (MPP) vendor was creating its own proprietary message passing API. In this scenario it was not possible to write a portable parallel application. MPI is intended to be a standard for message passing specifications that each MPP vendor would implement on its system. The MPP vendors need to be able to deliver high-performance and this became the focus of the MPI design. Given

this design focus, MPI is expected to always be faster than PVM on MPP hosts [15].

The first standard named MPI-1 contained the following features:

- A large set of point-to-point communication routines, by far the richest set of any library to date.
- A large set of collective communication routines for communication among groups of processes.
- A communication context that provides support for the design of safe parallel software libraries.
- The ability to specify communication topologies.
- The ability to create derived data types that describe messages of non-contiguous data.

MPI-1 users soon discovered that their applications were not portable across a network of workstations because there was no standard method to start MPI tasks on separate hosts. Different MPI implementations used different methods. In 1995 the MPI committee began meeting to design the MPI-2 specification to correct this problem and to add additional communication functions to MPI including:

- `MPI_SPAWN` functions to start MPI processes.
- One-sided communication functions such as `put` and `get`.
- `MPI_IO`.
- Language bindings for C++.

The MPI-2 specification was finished in June 1997. The MPI-2 document adds 200 functions to the 128 original functions specified in the MPI-1.

All the mentioned advantages have made MPI the standard for the future applications using message-passing services. The drawbacks relating dynamic process creation and interoperability are being successfully solved. In addition, the connectivity with the Globus system is an important feature that stresses the importance of MPI.

2.4 Globus

Globus is a new, extremely ambitious project to construct a comprehensive set of tools for building meta-computing applications [6]. The goal of the Globus project is to provide a basic set of tools that can be used to construct portable, high-performance services, which in turn support metacomputing applications. Globus thus builds upon and vastly extends the services provided by earlier systems such as PVM, MPI, and Legion [9]. The project is also concerned with

developing ways to allow high-level services to observe and guide the operation on the low-level mechanisms.

The toolkit modules execute on top of a meta-computer infrastructure and they are used to implement high-level services. The metacomputer infrastructure, or testbed, is realized by software that connects computers together. Two instances of such an infrastructure have been built by the Globus group. The first, the I-WAY networking experiment, was built in 1996; it connected 17 sites in North America and was used by 60 research groups to develop applications. The second metacomputer infrastructure, GUSTO (Globus Ubiquitous Supercomputing Testbed), was built in 1997 as a prototype for a computational grid consisting of about 15 sites. GUSTO in fact won a major award for advancing high performance distributed computing.

The Globus toolkit consists of several modules:

- *Communication module*: Provides efficient implementation of many of the communication mechanisms, including message passing, multicast, remote procedure call, and distributed shared memory. It is based on the Nexus communication library.
- *Resource location and allocation module*: Provides mechanisms that allow applications to specify their resource requirements, locate resources that meet those requirements, and acquire access to them.
- *Resource information module*: Provides a directory service that enables applications to obtain real-time information about the status and structure of the underlying metacomputer.
- *Authentication module*: Provides mechanisms that are used to validate the identity of users and resources. These mechanisms are in turn used as building blocks for services such as authorization.
- *Process creation module*: Initiates new computations, integrates them with ongoing computations, and manages termination.
- *Access module*: Provides high-speed remote access to persistent storage, databases, and parallel file systems. the module uses the mechanisms of the Nexus communication library.

The Globus toolkit modules are being used to help to implement high-level application services. One such service is what is called an Adaptive Wide Area Resource Environment (AWARE). It will contain an integrated set of services, including "metacomputing enabled" interfaces to an implementation of the MPI library, various programming languages, and tools for constructing virtual environments (CAVEs). The high-level services will also include those developed by others, including the Legion metacomputing system, and implementations of CORBA, the Common Object Request Broker Architecture. See [8] for more details on Globus, and the Globus Web site at `www.globus.org` for detailed information and current status of the project.

2.5 Java-RMI

The implementation of remote procedure calls (RPC) in Java is called Java-RMI [12]. The Remote Method Invocation in Java allows an application to use a remote service with the added advantages of being platform-independent and able to access to the rest of useful Java characteristics when dealing with distributed computing and the Internet in general.

The client/server model used by Java-RMI is however somewhat slow in the current implementations of Java, an especially important consideration when dealing with optimization algorithms. An additional drawback is that current trends in the communication markets seem leading to abandon the support for this model of computation.

2.6 CORBA, Active-X, and DCOM

Although many topics currently deal with multithreaded, parallel, and/or distributed programs, some higher-level research is devoted on how to glue together existing or future applications so they can work together in a distributed, Web-based environment. Software systems that provide this glue have come to create the term "middleware". CORBA, Active-X, and DCOM are three of the best known examples [17]. They and most other middleware systems are based on object oriented technologies. Common Object Request Broker Architecture (CORBA) is a collection of specifications and tools to solve problems of interoperatibility in distributed systems (`www.omg.org`). Active-X is a technology for combining Web applications such as browsers and Java applets with desktop services such as document processors and spreadsheets. The Distributed Component Object Model (DCOM) serves as a basis for remote communications, for example, between Active-X components (`www.activex.org`).

CORBA is especially important because it is growing in the application side rapidly; it allows clients to invoke operations on distributed objects without concern for object location, programming language, operating system, communication protocols, or hardware. Reusability, interoperatibility, load balancing, and the rest of mentioned advantages make CORBA a serious standard when dealing with objects in present distributed systems. In addition, the IIOP protocol allows WAN connections among distributed ORBs.

2.7 Others

There is an enormous number of other communication toolkits for constructing a new middleware system. Here, we only briefly review some of the most popular ones:

- **OpenMP**: OpenMP is a set of compiler directives and library routines that are used to express shared-memory parallelism (`www.openmp.org`). The OpenMP Application Program Interface (API) was developed by a group representing the major vendors of high-performance computing hardware and software. Fortran and C++ interfaces have been designed,

with some efforts to standardize them. The majority of the OpenMP interface is a set of compiler directives. The programmer adds these to a sequential program to tell the compiler what parts of the program to execute concurrently, and to specify synchronization points. The directives can be added incrementally, so OpenMP provides a path for parallelizing existing software. This contrasts with the Pthreads and MPI approaches, which are library routines that are linked with and called from a sequential program, and which require the programmer to manually divide up the computational work.

- **BSP**: The Bulk synchronous Parallel (BSP) model is a so-called bridging model that separates synchronization from communication and that incorporates the effects of a memory hierarchy and of message passing [18]. The BSP model has three components: processors, a communication network, and a mechanism for synchronizing all the processors at regular intervals. The parameters of the model are the number of processors, their speed, the cost of a communication, and the synchronization period. A BSP computation consists of a sequence of supersteps. In each superstep, every processor executes a computation that accesses its local memory and sends messages to other processors. The messages are requests to get a copy of (read) or to update (write) remote data. At the end of a superstep, the processors perform a barrier synchronization and then honor the requests they received during the superstep. The processors then proceed to the next superstep. In addition to being an interesting abstract model, BSP is now also a programming model supported by the Oxford Parallel Applications Center. More on BSP can be learned at `www.bsp-worldwide.org`.

- **Extensions to MPI**: While many hardware vendors have adopted the MPI standard and provide their own users with fast and stable implementations, there is no support for metacomputing with MPI for the moment being. PVM is designed to overcome that problem, but, since PVM is no longer the standard in the field most users have moved to MPI. To avoid changing the code of these users for metacomputing experiments PVMPI [5] has come to bridge the gap between PVM and MPI. But, in practice, this would require the user to substantially change his code. Only an interoperable MPI such as PACX-MPI [4] has finally provided access for metacomputing with MPI. A very promising wide-area implementation of MPI using the services in Nexus to interface Globus is the result of ongoing works [7].

- **Legion**: The Legion [9] research project at the University of Virginia (`www.virginia.edu`) aims to provide and architecture for designing and building system services that present the illusion of a single virtual machine. Persistence, security, improved response time, and greater throughput are among its many design goals. But, the key characteristic of the system is its ambition of presenting a transparent, single virtual machine

interface to the user. Legion aims at presenting a seamless computing environment with a single name-space, but supporting multiple programming languages (and models) and interoperability. It is an object-oriented system that attempts to exploit inheritance, reuse, and encapsulation; the distributed object programming system Mentat (a precursor to Legion) is in fact the basis for programming the first public release of Legion.

- **NetSolve**: A somewhat different, clientserverbased approach is adopted by NetSolve [2], a computational framework that allows users to access computational resources distributed across the network. NetSolve offers the ability to search for computational resources on a network, choose the best available resource based on a number of parameters, solve a problem (with retry for fault tolerance) and return the answer to the user. Resources used by NetSolve are computational servers that run on different hosts, and may provide both generic and specialized capabilities. The system provides a framework to allow these servers to be interfaced with virtually any numerical software. Access is achieved through a variety of interfaces; two which have been developed are as a MATLAB interface and a graphical Java interface. It is also possible to call NetSolve from C or FORTRAN programs using a NetSolve library API.

An interesting revision of heterogeneous distributed programming tools (including Globus, NetSolve, Harness, Legion, PVM, MPI, etc) can be found in (Sunderam and Geist 99).

2.8 Which of Them?

Since our goal is to construct an optimization system that spawns across several clusters of machines in a WAN environment we can identify the following important issues that must be met by the middleware:

- To have some abstract mechanisms to allow high-order communication operations by means of a rich programming interface.
- To be popular enough and extensively tested for becoming a good choice, not only at present but for future trends in communication models and in networking technology.
- To be general enough for highlevel programming but not too specialized in some kinds of applications, since our optimization problems can belong to quite different problem classes.
- To be targeted to a wide spectrum of applications and, at the same time, being able to provide efficiency in concrete clusters of machines.

Several other minor issues can be added to the previous list, but in practice the presented ones help in clarifying the decision. Since no one of the studied systems is by itself useful for our skeleton approach we need to select a communication toolkit for the implementation of our own middleware system. The

reason is that interfacing the middleware with the optimization skeletons is a specialized operation that is not present in any of the mentioned tools.

Although we have made a somewhat detailed presentation of many systems, only a few of them fit our requirements. First, the socket abstraction for message passing is powerful enough for our application domain, but it needs excessive low level manipulations when programming distributed systems with them. We discard other tools like Java-RMI since such a client/server model is still quite slow in present releases of Java; in addition, it seems that in the forthcoming years this standard is going to be abandoned in favor of other new models.

On the other side, CORBA seems the most widely accepted object broker for distributed systems. CORBA is a highly standard and cross-platform facility with the added advantage of being language-independent. However, it is still unclear if our skeleton engine will present a truly object-oriented interface to its environment (the same holds if Java were selected). This last, and the expected reduction in efficiency if the standard CORBA is chosen, both in a single cluster of machines and in a wide area network, are the reasons for not using it in the present implementation of the middleware for our applications.

We have reviewed also many other distributed programming paradigms such as BSP, OpenMP, and many tools for metacomputing such as Legion and others. None of them appear to be good candidates for our middleware implementation since their close relationship with well established types of applications (shared memory systems, virtual computers with easy use but with a difficult control to achieve efficiency, ...). NetSolve seems appropriate at first glance. It allows the user application to call remote procedures such as FFT, matrix operations, etc. with some blocking/non blocking options. However, this can be useful in some engineering computations only, because it does not allow to control processes nor hardware capabilities.

Then, our discussion is reduced to three communication tools, namely PVM, MPI, and Globus. We first discard Globus since it has an excessively ambitious spectrum of applications. Globus is quite complex for our needs: we need controlling processes and exchanging messages among workstations located in the same cluster or among geographically distributed clusters. However, we are not still able at present of evaluating the importance of Globus in parallel computing, since its popularity, efficiency, and controllability are continuously growing in these days. Nonetheless, it would be great that the selected underlying communications could be carried out with a toolkit having present or scheduled future interface with Globus.

Finally, we end with two systems, PVM and MPI. Although and heterogeneous computing system could be readily implemented on PVM, we choose MPI for several reasons. First, the message passing community has shifted from PVM to MPI naturally, due to the largest efficiency that can be got with MPI. Second, PVM is beginning to enter the "unsupported status", this being a serious drawback for future users and projects based in PVM. Third, the initial problems in MPI relating interoperability, dynamic process groups and some other minor details are being solved in MPI-2 and the new standards. Fourth, MPI has recently being used in many works to be explicitly extended for efficient

implementations in wide area networks. Fifth, MPI is quite popular, available, and standardized. Finally, MPI has already a direct extension to meet the Globus project through the Nexus services.

All these reasons lead us to rely on MPI for constructing the specialized middleware needed to be interfaced with the optimization skeletons for general and, at the same time, efficient distributed optimization. The next section will draw the main services needed for the middleware system and then a forthcoming section will deal with how it is planned to use this middleware from inside the skeleton implementations.

3 Middleware API: NetStream Description

This work is devoted to describe a C++ class containing basic and advanced services for message passing through a communication network. From now on, we will call this class `NetStream` since the design goals will lead us to define a "stream-like" interface for accessing the network.

Message passing is a well-known communication paradigm very useful in a set of assorted application domains, both for LAN and WAN services. PVM [14] and MPI [13] are two popular libraries that fit rather well this category. However, we envision some other goals that make this "raw" libraries appear working at too "low-level" for our target users. In fact, our start objectives for the communication library are listed below and own a close relationship to the necessities of the Spanish national funded project MALLBA (TIC1999-0754-C03-03):

- easy interface for people not being specialists in parallel programming,
- access to LAN as well as to WAN services,
- flexible and object oriented user interface,
- efficient message passing of objects through the network,
- easy extensibility with new services, and
- abstraction and re-utilization with a "light weight" presentation.

In order to cope with these goals the resulting system must show a great deal of concrete features. Since we need both basic and advanced services we need to define methods in the final C++ class devoted to these two types of users. In any case, we plan to offer methods having a very clear interface so that the learning time will be minimized. In addition, because we want to access both LAN and WAN characteristics an effort must be made to make a uniform interface for they two in terms of resulting methods of the class.

Besides that, efficiency is an important goal, given that we want to use the library both for sparse and intense message passing programs. And finally, we directly embrace the object oriented technology; the reason is that we really

want to separate implementation from conceptual services. Of course, abstraction, re-utilization, and extension must be taken into account because nowadays libraries continuously undergo revision steps in order to fix or add new services to the existing ones.

As a result, we adopt MPI [13] as the base communication library in order to implement `NetStream` because it is a standard in message passing and becauseof its efficiency and future connectivity with emerging technologies such as Globus [6]. However, this not prevent a future change in the implementation of the `NetStream` library services on a different underlying system. Also, we will use directly C++ as the base language since it is object oriented, very popular, and (at present) more efficient than Java implementations for the so many different kind of applications we are devising `NetStream`.

We will develop the whole library in a "stream-like" fashion. This means that we will only need to declare a `NetStream` object and then go on with it by invoking the appropriate methods. We will use the standard *inserter* `<<` and *extractor* `>>` operators in order to express reception and transmission of information on a net stream. This will bring uniformity to our new streams with respect to standard input/output streams and also it will allow the programmer input/output a sequence of objects in a single statement (as well as it helps in reducing the verbosity that would from using a named method instead of these operators).

```
NetStream netstream;
...
netstream << 9 << 'a' << "hello world";
...
```

Next subsection will deal with the definition of the basic services for novice users. Then, we will move on to more advanced services aimed at satisfying the needs of parallel programmers. After that, we will show and explain some examples of use. Finally, we will finish by summarizing the contents of this paper and discussion some open lines.

3.1 Basic Services in NetStream

Basic services are targeted to non-specialized users wanting an easy manner of sending and receiving information through the network. Consequently, these services will have clear semantics as well as an easy interface. Since there are a large variety of basic classes in the standard C++, we will overload the input/output methods for each of such basic classes. See the example below:

```
class NetStream
{
    public:

    NetStream ();              // Default constructor
                               // Constructor with source integer left unchanged
    NetStream (int, char **);  // Init the communications
    ~NetStream ();             // Default destructor

    void init(int,char**);     // Init the communication system. Invoke it only ONCE
    void finalize(void);       // Shutdown the communication system. Invoke it ONCE
```

```
// BASIC INPUT SERVICES                <comments>             BASIC OUTPUT SERVICES
// ============================================================================================================
   NetStream& operator>> (bool&   d);                          NetStream& operator<< (bool   d);
   NetStream& operator>> (char&   d);                          NetStream& operator<< (char   d);
   NetStream& operator>> (short&  d);                          NetStream& operator<< (short  d);
   NetStream& operator>> (int&    d);                          NetStream& operator<< (int    d);
   NetStream& operator>> (long&   d);                          NetStream& operator<< (long   d);
   NetStream& operator>> (float&  d);                          NetStream& operator<< (float  d);
   NetStream& operator>> (double& d);                          NetStream& operator<< (double d);
   NetStream& operator>> (char*   d);    /*NULL terminated*/    NetStream& operator<< (char*  d);
   NetStream& operator>> (void*   d);    /*NULL terminated*/    NetStream& operator<< (void*  d);
#ifdef _LEDA_
                    NetStream& operator>> (string&   d);                        NetStream& operator<< (const string&   d);
   template <class T> NetStream& operator>> (array<T>& d);      template <class T> NetStream& operator<< (const array<T>& d);
   template <class T> NetStream& operator>> (slist<T>& d);      template <class T> NetStream& operator<< (const slist<T>& d);
   template <class T> NetStream& operator>> (list<T>&  d);      template <class T> NetStream& operator<< (const list<T>&  d);
#endif

    int pnumber(void);                    // Returns the number of processes
    NetStream& _my_pid(int* pid);         // Returns the process ID of the calling process
    NetStream& _wait(const int stream_type);// Wait for an incoming message in the specified stream
    NetStream& _set_target(const int p);    // Stablish "p" as the default receiver
    NetStream& _get_target(int* p);         // Get into "p" the default receiver
    NetStream& _set_source(const int p);    // Stablish "p" as the default transmitter
    NetStream& _get_source(int* p);         // Get into "p" the default transmitter

   ...
};
```

Let us now describe the basic interface. First, the user must include the `netstream.hh` file in its program file. Then, he or she must declare a `NetStream` object. The constructor may have no arguments or else it might have the two arguments of the `main` program or the `init` method.

The user is supposed to make a `init()` call before all the system begin to work and a call to `finalize()` for shutting down the communication system (only once). In the middle of this parenthesis-like structure the user can input/output basic data types from/to the `NetStream` previously declared object. Normal classes such as `int`, `double`, and `char` are of course included among the large set of classes that can be exchanged through the net.

```
#include "netstream.hh"
int main (int argc, char** argv)
{
    NetStream netstream;
    int       mypid;
    ...
    netstream.init(argc,argv);  // Initialize the comm system
    netstream << set_target(1)  // Set the target process
              << set_source(1)  // Set the source process
              << my_pid(&mypid);// Get the pid of the calling process
    ...
    netstream << 9 << 'a' << "hello world"; // Send data through the net
    netstream >> i >> c >> str;             // Receive data from the net
    ...
    netstream.finalize();       // Shutdown the comm system
} // main
```

As you notice, before engaging in input/output operations the user can set/get the process number in the other end from/to which the communication is being achieved. For this purpose, four methods are readily provided (see

NetStream public interface). Notice that these, as well as other methods, begin with an underscore character "_". The reason is that the same methods exist for invocation inside and inserter << or extractor >> operator in a single sentence. This is included for compatibility with the *manipulator* philosophy of standard streams in C++. A manipulator is a method that can be fed into a >> or << operator in order to perform a task. A manipulator can be merged with standard input/output operations, thus providing a nice and uniform interface for streams. Manipulators can also have arguments; they look like normal methods with the exception that they can be used in isertions and extractions of a stream.

The method `pnumber()` allows the user to know the number of processes running in parallel, and the method `_my_pid()` returns as an argument the process identifier of the calling process in the set of parallel processes.

The method `wait()` allows the user to wait for an incoming message in a given stream. The most usual net stream the user will need is the `regular` stream for sending/receiving normal data to/from other processes.

```
netstream << wait(regular);  // Wait for a regular message
```

Finally, for these users having code that includes data types from the LEDA library, next versions of the `NetStream` class will support exchanging the LEDA types `string`, `array`, `slist`, and `list`.

After the user has typed is parallel program by using `NetStream`, he or she can compile it with the usual `mpicc` operating system command and the run it with the usual `mpirun` command.

3.2 Advanced Services in NetStream

There are several advanced services available for any `NetStream` object. These services are specially important for solving synchronization tasks, namely establishing synchonization points (called *barriers*), broadcasting one message to the rest of processes, and checking whether there is a pending message in the `regular` or `packed` stream. The corresponding methods in the `NetStream` class are (respectively) `_barrier()`, `_broadcast()`, and `_probe()`. As before, there exist methods with the same name and behavior that can be used as manipulators with the << and >> operators. See the following code to learn the syntax of the `NetStream` methods:

```
class NetStream
{
    public:
    ...                             // BASIC SERVICES already described
    NetStream& _pack_begin(void);   // Marks the beginning of a packed information
    NetStream& _pack_end(void);     // Marks the end of a packed and flush it to the net
    NetStream& _probe(const int stream_type, int& pending); // Check whether there are awaiting data
    NetStream& _broadcast(void);    // Broadcast a message to all the processes
    NetStream& _barrier(void);      // Sit and wait until all processes are in barrier
    ...
};
```

When programming for a LAN environment, passing basic C++ types such as `int` or `double` is OK with modern technologies, since the latency is low.

However, for a WAN environment sending many continuous messages with such basic types could provoke an unnecessary delay in communications. Network resources can be better exploited if the user define data *packets*.

Defining a data packet is very easy because only the manipulators

`pack_begin`

and

`pack_end`

must be used. All the output operations in between these two reserve words are put inside the same physical packet, with the ensuing savings in time. The contents of the packet are not forced to share the same base object class or type, thus improving the flexibility of this construction.

```
if(mypid==0)    // The sending process
{   ...
    strcpy(str,"this is sent inside a heterogeneous packet");
    netstream << pack_begin
                  << str << 9.9 << 'z'
              << pack_end;
    ...
}
else            // The receiving process
{
    ...
    netstream << wait(packed);  // Wait for a packed message
    netstream << pack_begin     // Reads the packed message
                  >> str >> d >> c
              << pack_end;
    ...
}
```

3.3 Future Extensions and Modifications of NetStream

Other, more sophisticated, services are planned to be added to `NetStream v1.0` in order to create an extended new version. besides, there are some improvements scheduled over the present version. Some of these improvements include:

- exchanging LEDA objects,
- defining the methods `init()` and `finalize` as static members, thus allowing the user to invoke them with a qualified scope call with the name of the class, instead of using one the the existing objects for this work; i.e. `NetStream::init()`.
- providing a default `NetStream` object meaning "the net",

- improving the implementation on top of MPI, in details like the maximum size of packets, and overloading some methods to be easily invoked as methods and not only as manipulators, and
- including some more services for flexible parallel data exchanges and synchronization.

This will result in a new version of the library with also enhanced methods to deal with LAN/WAN environments.

3.4 NetStream Example of Utilization

In this section we provide an example of utilization of some of the more interesting features of the `NetStream` class. We will include basic operations as well as other more sophisticated behavior such as sending/receiving packets for use in the WAN when the programmer judges inefficient to send basic (small) objects through a long distance connection. Also, some synchronization services such as creating a barrier or a wait operation are illustrated to shown the versatility of the library.

Notice that most of the methods are invoked inside the << and >> operators (what it is called stream *manipulators*) for the sake of uniformity and elegancy in C++.

```
#include "netstream.hh"
int main (int argc, char** argv)
{
    NetStream netstream;
    int       mypid;
    char      c;
    int       i, s, t;
    double    d;
    char      str[1000];

    netstream.init(argc,argv);      // Initialize the comm system
    netstream << my_pid(&mypid);    // Get the process identifier

    if (mypid==0)
    {
        strcpy(str,"hello world");
        netstream << set_target(1)  << set_source(1)
                  << get_target(&t) << get_source(&s)
                  << my_pid(&mypid);

        netstream << barrier;       // Synchronize

        netstream << 9 << 'a' << str;
        netstream >> i >>  c  >> str;

        cout << "process " << mypid << ":"
             << " sends to process " << t
             << " and gets data from processs " << s << endl
             << i << endl << c << endl << str << endl << flush;

        strcpy(str,"this is sent inside a heterogeneous packet");
        netstream << pack_begin
                  << str << 9.9 << 'z'
                  << pack_end;
    }
    else
    {
        netstream << set_source(0)  << set_target(0)
                  << get_source(&s) << get_target(&t)
                  << my_pid(&mypid);

        netstream << barrier;       // Synchronize

        netstream >> i >> c >> str;
        netstream << i << c << str; // ECHO

        netstream << wait(packed);  // Wait for a packed message
        netstream << pack_begin     // Reads the packed message
```

```
                >> str >> d >> c
                << pack_end;

        cout << "process " << mypid << ":"
             << " sends to process " << t
             << " and gets data from processs " << s << endl
             << str << endl << d << endl << c << endl << flush;
    }
    netstream.finalize();
}
```

4 Integration with the Optimization Skeletons

In this section we discuss an initial proposal for making the middleware system available for use in the optimization skeleton implementations. The middleware system is composed of several elements:

- The Application Program Interface (API).
- The System Files.
- The Instantiation Module (IM).

The API allows the skeletons to use the middleware. The system files contains information about the interconnected LANs, the processors, and the WAN as a whole. The instantiation module provides a way for the administrator to tune the behavior of the middleware for its own cluster.

We now proceed to explain their use in the following subsections.

4.1 The Application Program Interface (API)

The API for the skeletons to use the distributed system is defined by the set of data types and methods described in the previous section. By means of these data types and methods any skeleton will be able to spawn, change, terminate, and in general manage a complex system of parallel optimization skeletons.

As explained before, we give a C++ abstract interface to the user to send his/her objects through the network. This interface is very easy to learn because making an instance object and input/output values are directly supported in the basic communications module. This is good not only for experienced parallel programmers, but also for beginners wanting to send data to other processes in the network, whatever the target process (algorithm) might be.

All the skeletons, exact, heuristic, and hybrid must use the same API to access the network facilities. This will ensure an homogeneous implementation for all the parallel skeletons. The special needs of every kind of algorithm will be taken into account by accessing the flags and other indicators included to customize the well-known behavior of every primitive in the API.

Sequential skeletons will not use the API, unless a distributed optimization skeleton is requested to run on a single processor. Therefore, the middleware will be of interest for the development of distributed optimization skeletons.

This basic API includes simple services that will be combined in the future to yield more complex services. This layered approach is quite usual in communication protocols (e.g. OSI, TCP/IP, etc.). Some more complex services such

as automatic mapping of processes to processors, load balancing, and dynamic changes in the network of communicating skeletons will be addressed by combining the presented API services, and may be by extending their semantics or by adding new services in future versions of the middleware. Also, every skeleton developed within the MALLBA project is expected to be included among the predefined optimization skeleton classes.

4.2 The System Files

Since our distributed optimization system is intended to run on a cluster of machines in a second phase of the MALLBA project (after the initial core-sequential phase) we need a model of the local area network. The reason is that we do not only want a distributed skeleton system, but an *efficient* distributed skeleton system. This efficiency requirement clearly makes necessary to take into account the special characteristics of the LAN being used while preserving the abstraction and standardization of the API.

Our proposal consists in providing LAN and WAN information into some systems files. The instantiation module (IM) will help the user to refine some general values to obtain an improved model for its own communication network.

Internal models for most common networks (e.g. fast-ethernet, ATM, Mirynet) will be included for users having no idea on technical issues concerning their own network. At least, when installing the system, the kind of network to be used will be detected to get a good efficiency in most cases.

Since we are going to deal with very different LANs the LAN system file will contains the following data:

- **Name of the LAN**. Some default names will be provided, such as `ethernet`, `fast-ethernet`, `gigabit-ethernet`, `atm`, `mirynet`, `fddi`, `other`.
- **Theoretical bandwidth**. The default names will internally have an associate bandwidth depending on the official standards of the included networks, such as 155 Mbps for `atm` and 100 Mbps for `fast-ethernet`.
- **Number of processors**. The total number of processors to be used must be indicated beforehand. The processors to be used will be taken from a list of available processors by following the order in which this list is provided. Later in this section we explain how the processors will be managed.

The available set of processors needs to be defined by the user in order the system to know where to run the necessary optimization skeletons. Processors will be organized in clusters. Every cluster will contain thus a variable number of processors. For each processor the following information is needed:

- The **DNS name** of the processor.
- The **IP address** of the processor.
- The **clock speed**, if known.

- The **RAM memory**, if known.
- The **operating system** version, if known.
- Whether the processor is **dedicated** or not, if known.

At least, the name and/or the IP address must be given for every processor in order to be usable. The rest of parameters will have default values that can be tuned depending on the LAN knowledge the user has. The result of the LAN system file is a two-level hierarchy of machines and some additional info from which the API will take advantage to run more efficiently. The LAN file will validate at last some of the most critical values provided by the users, such as the name/address of every processor, in order to assess a minimum quality for the running environment.

As well as the LAN system file will help in a more efficient instantiation of the parallel optimization skeletons, a system file will exist to specify a model for the whole distributed system. After the definition of every LAN in the whole system, we can easily see that the distributed environment is considered as a set of machine clusters. At the geographically distributed level we specify some values concerning the whole set of computational resources to run a given skeleton.

The machines involved in the computations belong each one to one (and only one) given cluster. Thus, we need to know which clusters are considered and how they are linked. The considered parameters effecting the whole communication system are:

- A set of pairs indicating two interconnected clusters.
- For every pair of interconnected clusters a performance model will be given.

The performance model can be a `null` model, a `default` model, or a `user-defined` model. In any case the performance model will detail the following *weekly* information:

- A valid label indicating the *day of the week* (from `monday` to `sunday`).
- A beginning and terminating *day hour* `(0..23h)`.
- The expected *bandwidth* and *latency* available at this day and time range.

4.3 The Instantiation Module

The Instantiation Module (IM) is a tool providing the user with the ability to get as much knowledge as possible into the middleware system concerning processors, clusters, and WAN facilities.

The IM is the interface application that the user can invoke to instantiate and customize the automatic default behavior of the middleware. It is expected that the performance of the parallel optimization skeletons will raise as the middleware-hardware mapping is enhanced.

5 Concluding Remarks

In this first draft we have discussed the pro and cons of many communication toolkits from the point of view of the MALLBA necessities. Our conclusion is that MPI is the standard that better fits our needs. These needs can be briefly summarized in the following topics: availability, efficiency, and future trends. Some other candidates such as CORBA or Globus have been finally left out at this stage of the MALLBA project, although future enhancements of the services could lead to return back to consider (e.g.) CORBA for introducing load balancing, automatic process migration, and for overcoming the secure access necessities of nowadays servers.

With the LAN and WAN system files, the proposed API will provide a useful set of services to program distributed optimization algorithms, whatever the kind of skeleton is (exact, heuristic, or hybrid). More in dept study of this proposal will probably reveals that further tuning is needed, and this is the reason for presenting a layered architecture approach for the middleware API.

The `NetStream` library is the featured communication tool aimed at helping programmers of parallel program to exchange information through a network, whether LAN or WAN it might be. Also, two levels of utilization are possible, namely basic services for novice users and advanced ones for experienced programmers.

Abstraction, flexibility, and easy interface are some of the more important goals that influenced the design of `NetStream`. The interface and operations are continuously being improved resulting in new versions of this software.

References

[1] Andrews G.R.(2000) "Foundations of Multithreaded, Parallel, and Distributed Programming. *Addison-Wesley.*

[2] Casanova H, Dongarra J.J. (1995) "NetSolve: A Network Server for Solving Computational Science Problems". *TR CS-95-313*, Univ. of Tennessee (November).

[3] Comer D.E., Stevens D.L. (1993) "Internetworking with TCP/IP (Volume III)". *Prentice-Hall.*

[4] Eickermann Th., Henrichs J., Resch M., Stoy R., Völpel R. (1998) "Metacomputing in Gigabit Environments: Networks, Tools, and Applications". *Parallel Computing* 24:1847–1872.

[5] Fagg G.E., Dongarra J.J. (1996) "PVMPI: An Integration of the PVM and MPY Systems". Department of Computer Science, *TR CS-96-328*, Univ. of Tennessee.

[6] Foster I., Kesselmann C.(1997) "Globus: A Metacomputing Infrastructure Toolkit". *Int. Journal of Supercomputing Applications* 11(2):115–128.

[7] Foster I., Geisler J., Gropp W., Karonis N., Lusk E., Thiruvathukal G., Tuecke S. (1998) "Wide-Area Implementation of the Message Passing Interface". *Parallel Computing* 24:1735–1749.

[8] Foster I., Kesselmann C. (1999) "The Grid: Blueprint for a New Computing Infrastructure". *Morgan Kaufmann.*

[9] Grimshaw A.S., Wulf W.A. (1997) "The Legion Vision of a Worldwide Virtual Computer". *Communications of the ACM* 40(1):39–45.

[10] Gropp W., Lusk E. (?) "Why are PVM and MPI so Different?" *TR* Mathematics and Computer Science Division, Argonne National Laboratory.

[11] Haupt T., Akarsu E., Fox G. (2000) "WebFlow: a Framework for Web Based Metacomputing". *Future Generation Computer Systems* 16:445–451.

[12] JavaSoft (1997) "RMI: The JDK 1.1 Specification". `javasoft.com/products/jdk/1.1/docs/guide/rmi/index.html`.

[13] Message Passing Interface Forum (1994) "MPI: A Message-Passing Interface Standard". *International Journal of Supercomputer Applications* 8(3/4):165–414.

[14] Sunderam V.S. (1990) "PVM: A Framework for Parallel Distributed Computing". *Journal of Concurrency Practice and Experience* 2(4):315–339.

[15] Sunderam V.S., Geist G.A. (1999) "Heterogeneous Parallel and Distributed Computing". *Parallel Computing* 25:1699–1721.

[16] Tierney B., Johnston W., Lee J., Thompson M. (2000) "A Data Intensive Distributed Computing Architecture for "Grid" Applications". *Future Generation Computer Systems* 16:473–481.

[17] Umar A. (1997) "Object-Oriented Client/Server Internet Environments". *Prentice-Hall.*

[18] Valiant L.G. (1990) "A Bridging Model for Parallel Computation". *Communications of the ACM* 33(8):103–11.

[19] Womble D.E., Sudip S.D., Hendrickson B., Heroux M.A., Plimpton S.J., Tomkins J.L., Greenberg D.S. (1999) "Massively Parallel Computing: A Sandia Perspective". *Parallel Computing* 25:1853–1876.

Appendix: Public Interface of the NetStream Class

```
/*************************************************************************
 ***                              netstream.hh                         ***
 ***                                  v1.0                             ***
 ***                               November 2000                       ***
 ***                                                                   ***
 ***   Communication services for LAN/WAN use following the message    ***
 ***   passing paradigm.                                               ***
 ***                               STREAM C++ VERSION                  ***
 ***                               MPI implementation                  ***
 ***                            Developed by Enrique Alba              ***
 *************************************************************************/

#ifndef INC_netstream
#define INC_netstream

#include "mpi.h"
#include <assert.h>

//#define _LEDA_        // Uncomment for using LEDA
#ifdef _LEDA_
#include <LEDA/string.h>
#include <LEDA/array.h>
#include <LEDA/slist.h>
#include <LEDA/list.h>
#endif

// Class NetStream allows to define and use network streams trhough LAN and WAN

#define REGULAR_STREAM_TAG  0   // Used for tagging MPI regular messages
#define PACKED_STREAM_TAG   1   // Used for tagging MPI packet messages

#define NET_TYPE                MPI_Datatype    // Network allowable data types
#define NET_BOOL                MPI_CHAR    // Bools like chars
#define NET_CHAR                MPI_CHAR
#define NET_SHORT               MPI_SHORT
#define NET_INT                 MPI_INT
#define NET_LONG                MPI_LONG
#define NET_UNSIGNED_CHAR       MPI_UNSIGNED_CHAR
#define NET UNSIGNED_SHORT      MPI_UNSIGNED_SHORT
#define NET_UNSIGNED            MPI_UNSIGNED
#define NET_UNSIGNED_LONG       MPI_UNSIGNED_LONG
#define NET_FLOAT               MPI_FLOAT
#define NET_DOUBLE              MPI_DOUBLE
#define NET_LONG_DOUBLE         MPI_LONG_DOUBLE
#define NET_BYTE                MPI_BYTE
#define NET_PACKED              MPI_PACKED

#define MAX_MSG_LENGTH          10240       // Max length of a message
#define MAX_PACK_BUFFER_SIZE    10240       // Max length of a packed message

// Help structure for manipulators having one int& argument
class NetStream;
struct smanip1c     // "const int"
{   NetStream& (*f)(NetStream&, const int);             // The ONE argument function
    int i;                                              // The argument
    smanip1c( NetStream&(*ff)(NetStream&,const int), int ii) : f(ff), i(ii) {}  // Constuctor
};

struct smanip1      // "int*"   note: references do not work! "int&"
{   NetStream& (*f)(NetStream&, int*);                  // The ONE argument function
    int* i;                                             // The argument
    smanip1( NetStream&(*ff)(NetStream&, int*), int* ii) : f(ff), i(ii) {}  // Constuctor
};



// Tags for the available streams
const int any     = MPI_ANY_TAG;        // Tag value valid for any stream
const int regular = REGULAR_STREAM_TAG; // Tag value for regular stream of data
const int packed  = PACKED_STREAM_TAG;  // Tag value for packed stream of data


class NetStream
{
    public:

    NetStream ();               // Default constructor
                                // Constructor with source integer left unchanged
    NetStream (int, char **);   // Init the communications
    ~NetStream ();              // Default destructor

    void init(int,char**);      // Init the communication system. Invoke it only ONCE
    void finalize(void);        // Shutdown the communication system. Invoke it ONCE

// BASIC INPUT SERVICES                <comments>              BASIC OUTPUT SERVICES
// ============================================================================================================
   NetStream& operator>> (bool&   d);                          NetStream& operator<< (bool   d);
   NetStream& operator>> (char&   d);                          NetStream& operator<< (char   d);
```

```
   NetStream& operator>> (short&  d);                          NetStream& operator<< (short  d);
   NetStream& operator>> (int&    d);                          NetStream& operator<< (int    d);
   NetStream& operator>> (long&   d);                          NetStream& operator<< (long   d);
   NetStream& operator>> (float&  d);                          NetStream& operator<< (float  d);
   NetStream& operator>> (double& d);                          NetStream& operator<< (double d);
   NetStream& operator>> (char*   d);    /*NULL terminated*/    NetStream& operator<< (char*  d);
   NetStream& operator>> (void*   d);    /*NULL terminated*/    NetStream& operator<< (void*  d);
#ifdef _LEDA_
                      NetStream& operator>> (string&   d);                        NetStream& operator<< (const string&   d);
   template <class T> NetStream& operator>> (array<T>& d);     template <class T> NetStream& operator<< (const array<T>& d);
   template <class T> NetStream& operator>> (slist<T>& d);     template <class T> NetStream& operator<< (const slist<T>& d);
   template <class T> NetStream& operator>> (list<T>&  d);     template <class T> NetStream& operator<< (const list<T>&  d);
#endif

    int pnumber(void);      // Returns the number of processes
    bool broadcast;         // Determines whether the next sent message is for broadcasting

    // Input MANIPULATORS for modifying the behavior of the channel on the fly
    // NO ARGUMENTS
    NetStream& operator<< (NetStream& (*f)(NetStream& n)) { return f(*this); } // NO arguments

    NetStream& _barrier(void);  // Sit and wait until all processes are in barrier

    NetStream& _pack_begin(void);   // Marks the beginning of a packed information
    NetStream& _pack_end(void);     // Marks the end of a packed and flush it to the net
    NetStream& _probe(const int stream_type, int& pending); // Check whether there are awaiting data
    NetStream& _broadcast(void);    // Broadcast a message to all the processes

    // ONE ARGUMENT
    // "const int"
    NetStream& operator<< (smanip1c m) { return m.f((*this),m.i); }// ONE int& argument constant
    // "int*"
    NetStream& operator<< (smanip1 m)  { return m.f((*this),m.i); }// ONE int& argument

    NetStream& _my_pid(int* pid);           // Returns the process ID of the calling process
    NetStream& _wait(const int stream_type);// Wait for an incoming message in the specified stream
    NetStream& _set_target(const int p);    // Stablish "p" as the default receiver
    NetStream& _get_target(int* p);         // Get into "p" the default receiver
    NetStream& _set_source(const int p);    // Stablish "p" as the default transmitter
    NetStream& _get_source(int* p);         // Get into "p" the default transmitter

   private:
    bool system_up;                   // Is the comm system already running?
    int default_target, default_source; // Default process IDs to send-recv data to-from
    bool pack_in_progress;              // Defines whether a packet is being defined with "pack_begin-pack_end"
    int pack_index;                     // Index to be used for adding-extracting to-from a packed message
    char* pack_buffer;                  // Buffer to temporary storage of the packed being defined
    bool pack_in, pack_out;             // Define whether input-output packed message is being used

    void send(void* d, const int len, const NET_TYPE type, const int target);
    void rcv (void* d, const int len, const NET_TYPE type, const int source);
};

    // MANIPULATORS (must be static or non-member methods in C++ -mpiCC only allows non-member!-)
    // NO ARGUMENTS
    NetStream& barrier(NetStream& n);   // Sit and wait until all processes are in barrier
    NetStream& broadcast(NetStream& n); // Broadcast a message to all the processes
    NetStream& pack_begin(NetStream& n);// Marks the beginning of a packed information
    NetStream& pack_end(NetStream& n);  // Marks the end of a packed and flush it to the net

    // ONE ARGUMENT
        NetStream& __my_pid(NetStream& n, int* pid); // Returns the process ID of the calling process
        inline smanip1 my_pid(int* pid){ return smanip1(__my_pid,pid); } // manipulator

        NetStream& __wait(NetStream& n, const int stream_type);// Wait for an incoming message - helper
        inline smanip1c wait(const int stream_type){ return smanip1c(__wait,stream_type); } // manipulator

        NetStream& __set_target(NetStream& n, const int p); // Stablish "p" as the default receiver
        inline smanip1c set_target(const int p){ return smanip1c(__set_target,p); } // manipulator

        NetStream& __get_target(NetStream& n, int* p); // Get into "p" the default receiver
        inline smanip1 get_target(int* p){ return smanip1(__get_target,p); } // manipulator

        NetStream& __set_source(NetStream& n, const int p); // Stablish "p" as the default transmitter
        inline smanip1c set_source(const int p){ return smanip1c(__set_source,p); } // manipulator

        NetStream& __get_source(NetStream& n, int* p); // Get into "p" the default transmitter
        inline smanip1 get_source(int* p){ return smanip1(__get_source,p); } // manipulator

    // TWO ARGUMENTS - not used yet
    NetStream& probe(NetStream& n, const int stream_type, int& pending); // Check whether there are awaiting data


#endif
```