`NetStream`: a Flexible and Simple OOP Message Passing Service for LAN/WAN Utilization

Enrique Alba

November 9, 2001

Departamento de Lenguajes y Ciencias de la Computación
Universidad de Málaga
eat@lcc.uma.es

Abstract

This paper describes the design, goals, and details of `NetStream`, a C++ class containing services for communicating information through a network. The services in `NetStream` have been developed for running both in local as well as in wide area networks; therefore, it provides message passing in a manner that makes it useful for a large class of parallel programs. The services in `NetStream` have been developed as a layer on top of the MPI standard, although nothing prevents further implementations with a different system. These services are divided into two sub-classes, namely basic and advanced; this makes the resulting interface appropriate both for non-specialized persons and for experts in developing parallel programs.

1 Introduction

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 [3] and MPI [2] 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 [2] 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 [1]. 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 section will deal with the definition of the basic services for novice users in version 1.0. Then, we will move on to more advanced services in Section 3 aimed at satisfying the needs of parallel programmers. Section 4 details the differences among the sucessive versions of `NetStream`. In Section 5 we will show and explain some basic examples of use, just to arrive to Section 6 in which we include an example of how groups are dealt with. A performance analysis of times with parallel exchange of data with `NetStream` is shown in Section 7. Finally, we will finish by summarizing the contents of this paper and by discussing some open lines in Section 8.

2 Basic Services in `NetStream v1.0`

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). New versions of `NetStream` will define these methods as `static`, thus allowing a class scope invokation `NetStream::init()`. 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 were initially thought to support exchanging LEDA types such as `string`, `array`, `slist`, and `list`. However, at present, some changes in the availability of this data library and in the priorities of our project have leaded to not supporting LEDA in `NetStream`.

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 Advanced Services

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;
        ...
    }
```

4 Versions of `NetStream`

Some changes from version 1.0 has been already included to yield two new versions (1.5 and 1.6). `NetStream v1.5` extends v1.0 in several ways:

- Methods `init()` and `finalize()` are set to be static, thus being common to any instance of `NetStream` objects, and callable in a more intuitive way (e.g. `NetStream::init()`) in accordance to the global operations they perform for any object.
- Group management services are provided in the form of new methods, namely one method for obtaining/setting the default communicator, one method to create a group inside a given communicator and one method to link two communicators in order to send messages from one sub-group to the other.

```
// GROUP management
        // Set the netstream to a new communicator
void set_communicator(NET_Comm comm);
        // Get the present communicator in this netstream
NET_Comm get_communicator(void);
        // Create a new group inside the present communicator
static NET_Comm create_group(NET_Comm comm, int color, int key);
    // Create a bridge between local and remote MATCHING call
static NET_Comm create_inter_group(NET_Comm lcomm, int lrank,
                                   NET_Comm bcomm, int rrank,
                                   int strtrype);
```

- Do not consider LEDA objects anymore. From v1.6 on the library do not provide any support to send/receive LEDA objects.
- The method `int my_pid()` is added in order to have an easy invokation inside conditional and repetitive sentences (precedent version needs to invoke this through a stream-like sentence).

- This version supports `unsigned` and `long double` input/output through the net.

On the other hand, current version 1.6 adds an internal change allowing more efficient executions and eliminating some bugs of precedent version when using packets:

- Internal in/out independent buffers have been defined.

New versions of `NetStream` will address issues concerning WAN services for obtaining delay times of the packets *on-line*, in order to provide the user with the actual performance of the network. This will highly assist the library users in taking decisions on when and how send information to a far node in the WAN.

5 A Basic 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
    mypid = netstream.my_pid(); // Notice the new invokation in v1.5

    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();
}
```

6 Example of Groups Management

In this section we provide an example of utilization of the newly added methods to deal with groups of processes exchanging data in parallel. The methodology is simple: inside the present communicator a group of processes sharing the same *color* is defined and separated from the rest of processes. The communicator of the new group and the old group are explicitly linked by a method dealing with such inter-communicator matter.

```
#include <stream.h>
```

```
#include "stdio.h"
#include "../../netstream.hh"

int main(int argc,char ** argv) {
     NetStream netstream;

     char     msg[20];
     char     msg1[20];
     int      myrank, my_new_rank;
     int      local_root, remote_root, target;
     int      tag=99;
     NET_Comm new_comm, inter_communicator, my_comm;
     int      number_of_processes, half_size;
     int      color, key;

     NetStream::init(argc,argv);    // Init the comm system

            // Get the process ID of this process
     netstream << my_pid(&myrank);
            // Get the number of processes
     number_of_processes = netstream.pnumber();
            // Half the number of processes
     half_size           = number_of_processes/2;
            // The key does not need to be unique
            // nor starting at 0. It's useful for sorting
            // ranks inside new groups
     key                 = myrank;
     if (number_of_processes>=2)   // The first step is creating both groups.
    {
       if (myrank<half_size)  // First group is composed by processes 0..half_size-1
         color=0;   // Color shared by all the processes in the first  group
       else
         color=1;   // Color shared by all the processes in the second group

                // Get the communicator of the netstream
        my_comm  = netstream.get_communicator();
                // CREATE THE GROUPS
        new_comm = netstream.create_group(my_comm,color,key);
                // Set default communicator for I/O
        netstream.set_communicator(new_comm);
                // Find process ID in new group.
                // Notice that we invoke it as a method!
        my_new_rank = netstream.my_pid();

        cout << "\nProcess n:  " << my_new_rank << " group: "
             << color << " ...old process n: " << myrank << flush;

        // Do not forget to synchronize to begin communication!!!
        netstream << barrier;

    // Now we send a message to the last  process of the our group
    // and to the last process of the other group.
    if (color==0)
    {
            local_root  = 0;
            remote_root = half_size;
                    // Now we need an intercommunicator descriptor
```

```
            inter_communicator = netstream.create_inter_group
                        (new_comm,local_root,my_comm,remote_root,tag);
            if (my_new_rank==0)
            {
                strcpy(msg,"initial msg");
            target=half_size-1;
                cout      << "\nTarget process: "<< target << "\tSender process: "
                          << myrank << flush;
                netstream << set_target(target)  << set_source(0);
                netstream << msg;
            }
            else
            {
                if (my_new_rank==half_size-1)
                {
                    netstream << set_source(0)<< set_target(0);
                    netstream >> msg1;
                    cout <<"\n*Intramessage received: " << msg1 << flush;
                    strcpy(msg,"inter-msg");
                    netstream.set_communicator(inter_communicator);
                    target = number_of_processes-half_size-1;
                    cout << "\n*Intercomm target process: "<< target
                         << " Intercomm sender process: "  << my_new_rank << flush;
                    netstream << set_target(target) << set_source(0);
                    // The new communicator was selected as default before
                    netstream << msg;
                    cout << "\n***Intermsg sent: " << msg << flush;
                }
            }
    }
    else //The other group (color==1)
    {
        local_root  = 0;
        remote_root = 0;
            inter_communicator = netstream.create_inter_group
                        (new_comm,local_root,my_comm,remote_root,tag);
            if (my_new_rank==number_of_processes-half_size-1)
            {
                netstream.set_communicator(inter_communicator);
                // it is not necessary to modify the target attribute:
                netstream << set_target(0) << set_source(half_size-1);
                netstream >> msg1;
                cout << "\n***Intermessage received:" << msg1 << flush;
            }
    }
     }
     else
     {
        cout << "\nUnable to make groups. Number of processes smaller than 2."
             << flush;
     }
     NetStream::finalize();
} // main
```

7 Performance Evaluation

In this section we present some basic performance measurements to show that `NetStream` is usually as fast as a raw MPI program, with a slight overhead when sending packed data. The tests have been performed in a 100 Mbps Fast-Ethernet cluster. We analyze the exchanges between two stations in this cluster (Pentium III, 700 Mhz, 128 Mb RAM).

In Figure 1 we show the time in milliseconds of sending different amount of data of the basic `char`, `int` and `double` values. It can be observed that there is no overhead of `NetStream` over MPI for any amount of data.

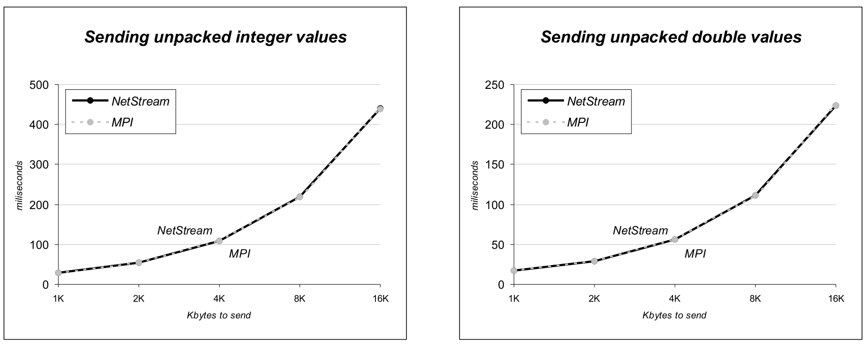

Figure 1: Sending `int/double` values between two workstations linked by Fast-Ethernet: MPI versus `NetStream` times.

In Figure 2 we show the trends for packets of different lengths of the basic data types `int` and `double` as they are very usual in numerical applications. A small overhead is then detected, especially for very long packets. The user must decide whether these small delays is worth-wile in the application at hands. In a WAN environment, and e.g. for optimization tasks, the packets are usually in the region of less than 2 Kb, and the overhead can be ignored.

8 Concluding Remarks

The `NetStream` library is a communication tool aimed at helping programmers of parallel program to exchange information through a network, whether LAN or WAN. 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.

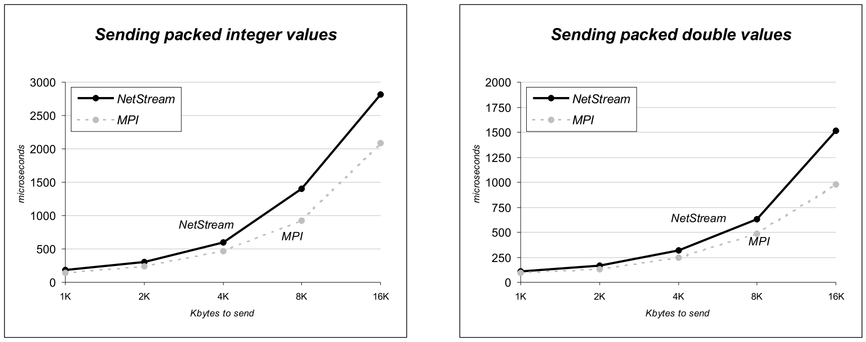

Figure 2: Sending packed `int/double` values between two workstations linked by Fast-Ethernet: MPI versus `NetStream` times.

Acknowledgements

This work is being partially funded by the Spanish Science and Technology Commission (CICYT), under contract TIC99-0754-C03-03.

References

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

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

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

Appendix: Public Interface of the NetStream Class

```
/**************************************************************************
 ***                           netstream.cc                             ***
 ***                         v1.6 - July 2001                           ***
 ***                                                                    ***
 ***                         v1.5 - March 2001                          ***
 ***                         v1.0 - November 2000                       ***
 ***                                                                    ***
 ***   v1.5 extends v1.0:                                               ***
 ***        .- Changes metods init() and finalize() to be static        ***
 ***        .- Incorporates process group management                    ***
 ***        .- Do not consider LEDA anymore                             ***
 ***        .- Contains a method "int my_pid()" for easy invokations    ***
 ***        .- Adds "unsigned" and "long double" input/output           ***
 ***                                                                    ***
 ***   v1.6 extends v1.5:                                               ***
 ***        .- Internal in/out buffers for packed separated             ***
 ***                                                                    ***
 ***   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>

// Class NetStream allows to define and use network streams through 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 NET_Comm            MPI_Comm

#define MAX_MSG_LENGTH              20480       // Max length of a message
#define MAX_PACK_BUFFER_SIZE        20480       // 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

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

    // GROUP management
    void set_communicator(NET_Comm comm);     // Set the netstream to a new communicator
    NET_Comm get_communicator(void);          // Get the present communicator in this netstream
    static NET_Comm create_group(NET_Comm comm, int color, int key); // Create a new group inside the present communicator
        // Create a bridge between local and remote MATCHING call
```

```
    static NET_Comm create_inter_group(NET_Comm lcomm, int lrank, NET_Comm bcomm, int rrank, int strtrype);


// 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);

   // Extended data types from version 1.5 on
   NetStream& operator>> (unsigned char&       d);              NetStream& operator<< (unsigned char       d);
   NetStream& operator>> (unsigned short int&  d);              NetStream& operator<< (unsigned short int  d);
   NetStream& operator>> (unsigned int&        d);              NetStream& operator<< (unsigned int        d);
   NetStream& operator>> (unsigned long int&   d);              NetStream& operator<< (unsigned long int   d);
   NetStream& operator>> (long double&         d);              NetStream& operator<< (long double         d);


    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

    // BASIC CLASS METHODS FOR MANIPULATORS
    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);    // Establish "p" as the default receiver
        NetStream& _get_target(int* p);         // Get into "p" the default receiver
    NetStream& _set_source(const int p);        // Establish "p" as the default transmitter
    NetStream& _get_source(int* p);             // Get into "p" the default transmitter

    // AUXILIAR PUBLIC METHODS FOR ALLOWING EASY MANAGEMENTS OF NETSTREAMS
    int my_pid(void);                   // Returns the process ID of the calling process

   private:
    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 packin_index;                   // Index to be used for extracting from a  IN  packed message  - v1.6
    int packout_index;                  // Index to be used for adding     to   an OUT packed message  - v1.6
    int pending_input_packet;           // Is there a pending packet already read into the IN buffer?  - v1.6
    char* packin_buffer;                // Buffer to temporary storage of the IN  packed being defined - v1.6
    char* packout_buffer;               // Buffer to temporary storage of the OUT packed being defined - v1.6
    bool pack_in, pack_out;             // Define whether input-output packed message is being used
    void reset(void);                   // Reset member variables of this class
    NET_Comm my_communicator;           // Communicator of this netstream

        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);

}; // class NetStream

    // 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
```