Accepted and presented in
Euromicro Parallel, Distributed and Network Processing PDP 2002

Modular MPI Components and the Composition of Grid Applications

J.Y.Cotronis
Dept. of Informatics and Communications, Univ. of Athens, Panepistimiopolis, 157 84, Greece
cotronis@di.uoa.gr Tel. + 301 7275223 Fax + 301 7275214

Abstract

The Ensemble methodology supports the design and implementation of message passing applications, particularly MPMD and those demanding irregular or partially regular process topologies. In Ensemble applications are built by composition of modular message passing components. We outline the Ensemble Software Architecture (ESA) and give an overview of the concepts and its supporting tools. We present extensions of ensemble components for composing Grid applications and outline their transformation to pure MPI executables and their execution on MPICH-G2. We demonstrate by building two simple applications, one SPMD and one MPMD where the former SPMD code is reused.

1. Introduction

We have developed the Ensemble methodology [1,2,3,4] for designing and building message passing (MP) applications based on modular MP components and composition. We have developed tools for designing and implementing components, designing topologies, specifying allocation resources and generating composition directives. These tools comprise the Ensemble Software Architecture (ESA), which has been developed on top of PVM [8] and MPI [10,11], the two most popular APIs. In addition to the general benefits of modular design, Ensemble overcomes three design and implementation difficulties.

The first is that implementation of MP applications does not only depend on the application design, but also on the target MP API, mainly because of the process model each API adopts. Some process topologies are easier to establish than others on specific APIs. For example, it is easy to create tree topologies (regular or irregular) in PVM and regular ring or grid topologies in MPI, but more difficult the other way round. Topologies not well suited to an API may certainly be created, but require specialized programming. Ensemble hides these idiosyncrasies of APIs and implementations maintain the original design.

The second difficulty is that APIs favour regular topologies and do not adequately support irregular or partially regular ones. Irregular topologies may be derived either from irregular domain decomposition and/or from functional decomposition [5]. They may give better performance, but they are much more difficult to design and implement than regular SPMD designs, which is the “favourite” model. In SPMD all processes are spawned from the same executable, but it is also implicitly assumed that they form regular topologies, usually a two dimensional mesh. Process topologies are established by implicit communication channels, expressed by symmetric calls of send and receive operations. For regular topologies the designer develops topology functions, which given a process identifier (e.g. rank) they return the identifiers of its communicating processes. These functions are usually parameterised to return the identifiers of processes in any size of the regular topology. For topologies, which are not SPMD and not globally regular but only locally or even altogether irregular, general functions cannot be derived and consequently ad hoc programming methods are used. Ensemble provides support for designing and implementing such applications.

The third difficulty is that modularity of MP components is limited [5]. The task a designer of a message passing program faces is to express in a source program P the interactions of all processes, which will be spawned from the executable of P, in all possible positions in the topology and for any size of the topology. Modularity is limited by the use of specific process identifiers (e.g. rank) or topology functions in send or receive calls, which presuppose a specific regular topology. Ensemble supports the design and implementation of modular MP components, which may be used in any topology, whether regular, partially regular or irregular.

In this paper we extend Ensemble to enable implementations to run on Grids [6] and particularly on MPICH-G2 [7,12]. We use the Globus RSL [9] language for application composition. Grids impose additional requirements for program modularity, since

applications may share components developed by different teams. An application may still be required to run independently, possibly as an SPMD, but it may be also required to co-operate with other applications, running together as MPMD. Such applications cannot in general be implemented in MPICH for two inter-related reasons a) the mpirun command may spawn processes from different executables, but the same command line arguments are passed to all of them, and b) MPI standard specifies that the developer should not assume a specific order for rank creation. Consequently, in an MPMD program, processes may find their rank, but they cannot even determine their neighbours. Previous Ensemble implementations support the design and development of MPMD programs under the assumption that process ranks are given in the order of their appearance in the process group file. In the grid implementation we have alleviated this assumption as the Globus RSL language permits each process to have its own command line arguments.

Previous Ensemble implementations were trying to abstract from particular APIs (PVM, MPI) and consequently program components could only use a limited semantically common subset of routines (e.g. send, receive, broadcast). In this paper we define MPI modular components and their composition supporting all point-point as well as collective MPI communication.

The structure of the paper is as follows: In section 2, we overview ESA; in section 3 we present the Ensemble MPI modular components and their transparent transformation to pure MPI sources and executables; in section 4 we outline topology design of two applications; in section 5 we present the generation of RSL composition scripts and the execution of applications; finally in section 6 we present our conclusions and plans for future work.

2. Ensemble Overview

Ensemble specifies a software architecture common for all MP applications in any API (figure 1). Differences in APIs are hidden in the ESA tools. The Ensemble software architecture (ESA) is divided in two layers: the Abstract Design and Implementation (AD&I), which is the responsibility of the programmer and the Architecture Specific Implementation (ASI), in this case MPICH, which is generated from the AD&I and is transparent to the developer. In the AD&I the programmer develops a complete, but abstract MP implementation, which is transformed into an ASI on the target execution environment (cluster, MPP or Grid).

2.1 Abstract Design and Implementation

The AD&I consists of three well-separated implementation parts. Two of them, namely the virtual components and the symbolic application topology are independent of the execution environment. The third one, the resource allocation is the bridge between the AD&I and the execution environment.

2.1.1 Virtual Components. A virtual component is an implementation abstraction of a MP program and consists of three attributes: the envelope, the arguments and the source code.

The first attribute, the envelope, is an abstraction of envelope related data in MPI calls. The envelope specifies abstract names for contexts that a component uses and within them abstract roots for collective calls and abstract point-point interaction. For point-point communication, ports are introduced, which is an abstraction of the envelope triplet (context, rank, message tag). Virtual ports with the same semantics are treated as an array of ports (MultiPort). The virtual envelop reflects the fact that MPI calls use four argument types, which determine envelopes; Context (communicator) and within it Ranks, Message Tags, and Roots (Rank). All four are specified in a component's virtual envelope.

The second component attribute, the arguments, correspond to command line arguments passed upon process spawning; they are distinguished in application and topology arguments. Application arguments are determined by the application requirements (e.g. I/O data files) and topology arguments (usually integers) determined by the distributed algorithms requirements, which reflect some measure of the topology (e.g. size of a ring topology). They are distinguished for semantic reasons.

Finally, the third attribute is the MPI source code in C. This code looks like an MPI program with one exception. All envelope-related arguments in MPI communication and synchronization calls refer to the virtual envelope. All other arguments have the usual bindings. For point-point communications the code refers to envelope ports.

Virtual Components are the heart of the Ensemble methodology. At compile time (pre-processing) all virtual envelop names are replaced by appropriate MPI envelop bindings and at process spawn time envelope are given actual values for Contexts, ranks, message tags and roots. A component may for example specify a virtual group (virtual context). All processes eventually spawned from this component must belong to some group (context). The actual

group is not known at compile time, but will be determined at run time by appropriate command line arguments. The group may involve processes spawned from the same component (SPMD) or from different components (MPMD). Also processes spawned from the same component may belong to different groups, all having the same virtual context, but associated with different actual contexts.

Ensemble Design & Implementation

Abstract Design&Implementation (AD&I)

Virtual Components
Envelope
Contexts
MultiPorts-ports
Roots
Arguments
Topology (Top-Args)
Application (Appl-Args)
Source Code
MPI Code with macros referring to Envelope names

Symbolic Topology
Processes
Instantiation
Interface
Top- Args
Interaction
Groups
Point-Point
Roots

Resource Allocation
Mapping
Appl-Args
Location of sources and executables

Arch.+.MPI Implementation

Generation of
MPI Source Codes and Make files
Composition Script
Binaries
procgroup or RSL
Mpirun

EXECUTION ENVIRONMENT

Fig. 1: The Ensemble Software Architecture

2.1.2 The Symbolic Topology. It is an abstraction of a process topology, which specifies the number of processes required from each component, each process's interface and its interaction with other processes. For each process the programmer specifies its actual envelope in a symbolic form. If the envelope specifies a group, then it is associated with a symbolic name. All processes associated with the same symbolic name will belong to the same group. For point-point communication AD&I resembles the task/channel model [5], but in a two-step manner: step one, within component code (task to port); and step 2 specified in the topology (port-port binding) outside processes. In a way we have extended the task/channel model for collective communications. Virtual groups are defined within components, which are associated with symbolic names in the topology.

2.1.3 Resource Allocation. The Ensemble design obtained in the first two parts is abstract. On the one hand it is independent of any execution environment, but on the other it other cannot "run" as it is. In AD&I we also specify the mapping of processes, as well as the location of source and executable files, input and output files in the execution environment.

2.2 Architecture Specific MPI Implementation

An architecture specific MPI implementation is transparently generated from an AD&I. It comprises of pure MPI sources (together with make files) and a composition script. MPI source files are generated from Ensemble components, which are compiled into modular MPI components. For single domain systems (e.g. clusters or MPPs) the composition script is in procgroup format is used and for Grids in a Globus RSL format. Each line of procgroup or each job request in RSL specifies the spawning of a single process, as each process has distinct arguments determining its communication bindings (actual point-point and collective communications).

In the sequel we demonstrate the development of applications by the Ensemble Methodology. We develop two solutions to a simple problem: There are processes, called terminal, which get an integer argument and require the maximum.

We describe the AD&I which is the programmer's responsibility and outline the generation and execution of "pure" MPI programs. The first implementation is an SPMD program of terminal processes. The second is an MPMD, in which terminal processes are grouped with server processes; servers find the local minimum within their group and cooperate (in a ring fashion) to find max which then broadcast to their terminals.

In the next section we develop the two Ensemble components (terminal and server) and outline their transformation into MPI code. In section 4 we design the two implementations using these two components. In section 5 we follow their composition and execution.

3. Developing Modular Components

We develop virtual components for terminal and server and outline their transformation into "pure" MPI executables.

3.1. The Virtual Components

We specify their envelope, arguments and code for terminal and server.

3.1.1 Terminal Component. Terminal processes get their integer argument, and then call MPI_Reduce with MPI_MAX specifying a context and a reduction root. Finally they call MPI_Bcast by which the root broadcasts the maximum. The virtual envelope is

Virtual Envelope of Terminal Component
Context1: LocalGroup /* context for group ops */ *Root1*: CalcRoot /* calculates Max */

Terminal has only one application argument, namely int-val its integer and no topology arguments.

Arguments of Terminal Component
Appl-Arg: int-val /* the integer argument */

The terminal code calls a procedure SetEnvArgs immediately after MPI_Init which must be considered as important. SetEnvArgs parses binding communication data in argv and assigns values to envelope data. ParseArg is a utility function, which selects in command line arguments (argv) a value preceded (indexed) by “int-val”, the name of the application argument and puts its value in variable Val. The code uses MPI Reduce and Broadcast routines, but envelop parameters for Root and Communicator are refer to virtual envelope names, i.e. CalcRoot and LocalGroup by a macro ENVRoot(CalcRoot, LocalGroup).

Code of Terminal Component

```
main (int argc, char **argv)
{ Int GlobalMax, Val;
  MPI_Init (&argc, &argv)
  SetEnvArgs(&argc, &argv); /* set envelop */
  ParseArg(int-val, Val); /* parse argument*/
  MPI_Reduce(&Val,&Max,1,MPI_INT,MPI_MAX,
     ENVRoot(CalcRoot,LocalGroup)); /*find Max*/
  MPI_Bcast(&GlobalMax,1, MPI_INT,
     ENVRoot(CalcRoot,LocalGroup)); /*bcast Max */
MPI_Finalize ( ); }
```

The actual Communicator and the actual rank of the Root executing the reduction and broadcast are not specified. This code generally specifies that a root process CalcRoot in a group LocalGroup will reduce Max and will broadcast it to the other processes in the group. The actual group and the actual root will be specified in the Topology part of the application, outside the components themselves. Each process will be passed appropriate arguments for constructing the communicator and the rank of the root. In the SPMD solution, the all-terminal solution, all terminals will be in the same group, and one of them will make the reduction (it could be any of them, which one will be determined in the topology). In the MPMD solution the terminals will be organized in different groups, each having a server as the reduction root.

3.1.2 Server Component. Server processes parse their Rsize (Ring size) argument, and then call MPI_Reduce with MPI_MAX within TerminalGroup and find the local Maximum. Within ServerGroup, they repeatedly send their current max to Out port, receive a value from In port and if greater than max keeps it as max. Finally they call MPI_Bcast by which the root sends the maximum to all processes of TerminalGroup. Its virtual envelope is

Virtual envelope of Server Component
Context1: TerminalGroup /* terminals and server */ *Root1*: CalcRoot /*calculates Local Max */ *Context2*: ServerGroup /* context of Servers*/ *Port1*: Out[1..1]/* output port */ *Port2*: In[1..1] /* input port */

The range 1..1 denotes that there is exactly one port for Out and one for In.

Server has only one topology argument, the size of the server ring.

Arguments of Server Component
Top-Arg: RingSize /* the size of server ring */

In server code we have used MPI routines with the same bindings as MPI routines, except where communication processes or groups are required, in which case they refer to Virtual envelope names for contexts, roots and ports.

All envelop arguments of point-point communication (Rank, Message Tag, Communicator) refer to virtual envelope names by a macro i.e. ENVPort(Out, 1, Servers). This macro denotes that a message is to be sent to port 1 (here the only one) of Multiport Out within the Servers group. There is a third macro ENVComm(VComm), which does not appear in terminal or server implementations and refers to the communicator of a context, which may be used in other MPI calls (probe, wait, etc.). With these three simple macros MPI code may refer to virtual envelope names and by expansion generate appropriate MPI bindings. Macros ENVPort and ENVRoot rely on the consecutive appearance of envelope related arguments in MPI calls (Root and Communicator) and (Rank, Message Tag and Communicator). We have experimented with other macros, but these proved the most simple and convenient. We believe that such code is straightforward to develop and does not deviate much from pure MPI calls.

Virtual Code of Server Component

```
main (int argc, char **argv)
{ Int Max, Temp, d=INT_MIN, Rsize, I;
 MPI_Status status;

 MPI_Init (&argc, &argv);
 SetEnvArgs(&argc, &argv);
 ParseArg(RingSize, Rsize);

 MPI_Reduce(&d, &Max, 1, MPI_INT, MPI_MAX,
     ENVRoot(CalcRoot,TerminalGroup));

 For (I=1; I<Rsize; I++) {/* Rsize –1 cycles */
 MPI_Send(&Max, 1, MPI_INT,
      ENVPort(Out,1,Servers));
 MPI_Recv(&Temp, 1, MPI_INT,
      ENVPort(In,1,Servers), status);
 If (Max < Temp) then Max=Temp;};/* end loop */

 MPI_Bcast(&Max, 1, MPI_INT,
   ENVRoot(CalcRoot, TerminalGroup));
 MPI_Finalize ( ); }
```

We note, that the code does not use functions for determining the next ((Rank+1) mod Rsize) and previous ((Rank-1) mod Rsize) neighbours using the process rank, which is a technique applicable for regular topologies. Although, this practice is not prohibited in Ensemble (it couldn't be anyway) it is not recommended as it restricts modular designs. For example, server processes could not be connected in disjoint ring topologies, which is possible in the above with the appropriate bindings for Out[1] and In[1] ports.

3.2 Generation of pure MPI code

From the Virtual Envelope and Code of components we generate pure MPI executables. A central element in this transformation is a structure called EnvArgs. The Virtual Component Code is wrapped by the declaration of EnvArgs and SetEnvArgs. EnvArgs is used for storing the actual MPI envelope data required by each process. Its top down declaration is shown in table 1. EnvArgs is an array, each element of which stores envelope data of one context (communicator). NrContexts is the number of contexts, as specified in the Virtual Envelope of a component. Each context element keeps envelope data for its Communicator and Process Rank, as well as for its Roots (namely ranks) and MultiPorts (a port is a pair rank and message tag) in two arrays. Procedure SetEnvArgs, which is called in each process after MPI_Init, parses appropriate argv values and performs the assignments to EnvArgs fields.

Table 1: Declaration of Structure EnvArgs

```
Context EnvArgs[NrContexts];

typedef struct
{      MPI_Comm ActualComm;
       int MyRank;
       int NrMPorts;
       MPortType MPorts[GlNrMPorts+1];
       int NrRoots;
       RootType Roots[GlNrRoots+1];
} Context;

typedef struct
{      int NrPorts;
       PortType Ports[GlNrPorts+1];
} MPortType;

typedef struct
       int Rank;
       int MessageTag;
} PortType;

typedef struct{int Root;} RootType;
```

The Virtual Code is also wrapped by macro definitions for ENVRoot, ENVPort and ENVComm and constant definitions by which MPI virtual routines are transformed to pure MPI routines, with proper envelop bindings to the elements of EnvArgs. For example the reduce call in the terminal source

```
MPI_Reduce(&Val, &Max, 1, MPI_INT, MPI_MAX,
  ENVRoot(CalcRoot, LocalGroup));
```

expands to

```
MPI_Reduce(&Val, &Max, 1, MPI_INT, MPI_MAX,
  EnvArgs[1].Roots[0].Root),
  EnvArgs[1].ActualComm);
```

Where EnvArgs[1] denotes the Context of LocalGroup and Roots[0] the CalcRoot.

Ensemble components are now transformed into pure MPI code and all calls have their proper bindings.

An important consideration is performance. The only overhead imposed to the execution of the generated code is the execution of SetEnvArgs, which is negligible. We took care so that MPI calls do not have any run time overhead; although the number of Roots and MultiPorts are in general different in each Context, we have used an array (bounded by the highest value of number of Roots and MultiPorts, respectively in all contexts of a component) rather than using a dynamic structure, which would not "waste" memory space. The reason for not using a dynamic structure is that during execution time each communication call would need one or two indirect memory accesses, which would reduce application performance. Using arrays all envelope-related data is bound at compile time. The "wasted" space is insignificant. The true value of Roots and MultiPorts in each Context and the true value of Ports in each

MultiPort are stored in NrRoots, NrMultiPorts and NrPorts respectively.

We have also defined (not shown here) a structure called SymbolicName for keeping all symbolic process names for processes used in the AD&I (e.g. terminal[1]); these names may be used in printf statements for symbolic program tracing and debugging.

4. Designing Symbolic Application Topologies

Having developed terminal and server components we proceed with developing application topologies.

4.1 All Terminals AD&I

In this design (fig. 2) we depict six processes from component terminal. All terminals belong to the same group and one is assigned to be the root. We associate their virtual LocalGroup with the symbolic group name TermG.

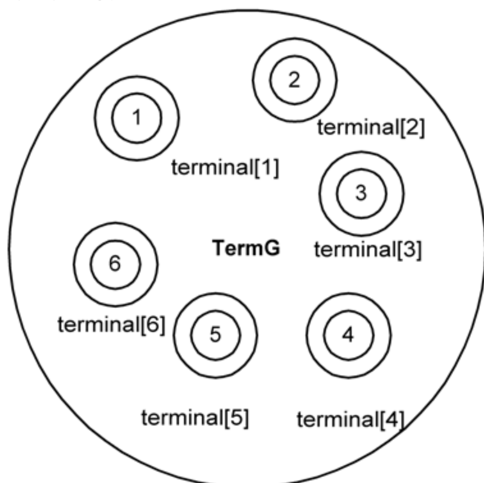

Fig. 2: Six terminal processes in TermG

4.2 Terminal Servers AD&I

In the second design (fig. 3) we use three server processes. Server terminal groups are of different sizes. Server[1] is grouped together with terminal[1] by associating TerminalGroup of server[1] and LocalGroup of terminal[1] with symbolic group name TG1. Similarly server[2] is grouped with terminal[2] and terminal[3] by associating TerminalGroup of server[2] and LocalGroup of terminal[2] and terminal [3] with symbolic group name TG2. Finally server[3] is grouped with terminal[4], terminal[5] and terminal[6] by associating TerminalGroup of server[3] and LocalGroup of terminal[4], terminal[5] and terminal[6] with TG3. Furthermore, server processes are grouped together by associating their ServerGroup with symbolic name Servers.

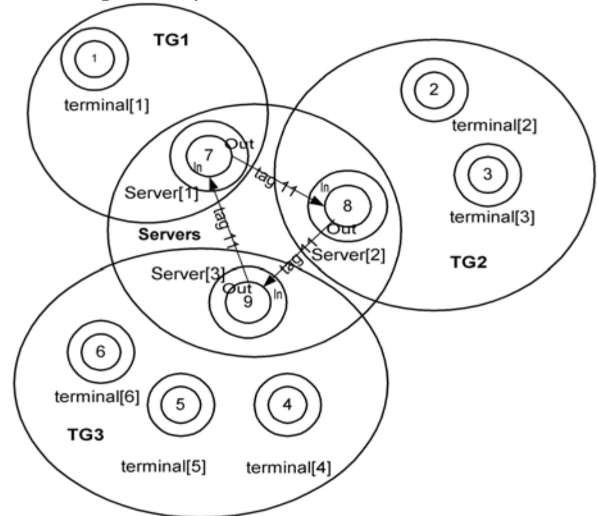

Fig. 3: Six terminals and three servers in groups

This design is depicted in figure 4 as a screen dump of our design tool, called Graphical Ensemble Tool (GrEnT), which supports Ensemble AD&I Design and the generation of composition scripts.

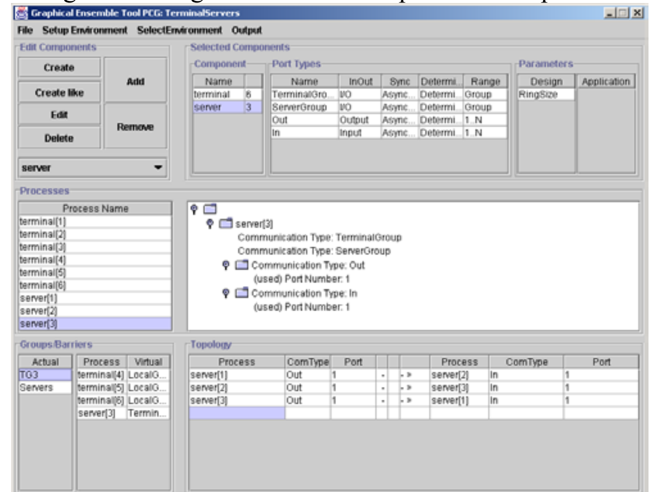

Fig 4.: GrEnT design for Terminals and Servers

In the screen dump we see on the top the two components, terminal and server. The virtual envelope and argument of the marked server are displayed. In the middle left panel the names of the six terminals and three servers are displayed (server[3] is selected). In the large window in the middle the interface of the selected server[3] are displayed. In the bottom left the symbolic group names in which server[3] belongs and a list with the other members in the TG3 group. Finally, in the large window in the bottom, the server ring point-point connections are displayed.

5. Composition Script and Application Execution

The RSL composition script is produced from AD&I by generating one request for each process, specifying the machine on which the executable will be spawned, the environment, the default directory, the executable and of course the arguments. The request for terminal[1] of the all terminal design is

```
(&(resourceManagerContact="gtest1.di.uoa.gr")
 (count=1)
 (label="subjob 0")
 (environment=(GLOBUS_DUROC_SUBJOB_INDEX 0))
 (arguments=1 terminal 1 1 0 MPI_WORLD 0 0 1
       TermG 0 1 1 terminal 1 int-val 134)
 (directory="/home/Ensemble1")
 (executable="terminal")
)
```

The argument list, which is processed by SetEnvArgs to set EnvArgs requires more detailed explanation.

5.1 The Arguments

The first argument of each process is an integer, internally generated, uniquely identifying the process. We use these ids to define point-point communication and roots. SetEnvArgs replaces ids with ranks (see 5.2). The second and third are symbolic name and index in AD&I. Then an integer indicates how many splits of MPI_WORLD_COMM will be performed. In the all terminal design, only one split corresponding to the construction of LocalGroup. Then envelope information for MPI_COMM_WORLD follows (always present). By convention we associate with it color 0 and symbolic name MPI_World; there are no Multiports and no Roots in this context. Then information for the first (and last in this case) split follows; the color (1) and the symbolic group name used in AD&I (TermG). According to MPI all processes in a group must call split routine. The number of Multiports follows, in this case 0, and the number of roots, in this case 1. Then envelope data for the root, its unique id, and symbolic name (terminal[1]). The arguments that follow are processed by ParseArgs and are pairs of virtual argument names and values.

Most terminal arguments are the same. The differences are in their unique identification (id and symbolic names) and the value of the integer. This is a simple SPMD design and the advantages cannot yet be demonstrated, except that the reduction root can be externally selected and not fixed in the code.

In the terminal server design the Ensemble advantages become apparent. There are significant differences in the arguments of terminals. As the six terminal processes are grouped in three disjoint groups, namely TG1, TG2 and TG3, a different color is used in each case; also each group has a different root. TG1 is associated with color 1 and its root is id=7 (server[1]), which also is part of TG1. Similarly, TG2 is associated with color 2 and its root is id=8 (server[2]), which is also part of TG2. Similarly for TG3. The three groups are created by the same collective split calls, as groups are disjoint.

Server processes participate in another group with color 4 and symbolic name "Servers". Terminal processes do not participate in it and are given –1 as split color. If color > 0 the process will be a member of the group of the constructed communicator. If color < 0 split routine is called with MPI color set to MPI_UNDEFINED returning MPI_COMM_NULL. In the former case the communicator is stored in array EnvPars, whilst in the latter it is ignored. The context associated with Servers has two Multiports each having one port. Information follows for each port (id, symbolic name and message tag). There are no roots in Context servers. Finally, argument tag RingSize and value 3 completes the argument list of server processes.

Let us point out that by passing a different set of arguments to processes terminal and servers they may be grouped into different configurations. Terminal[1] for example would join TG2 by changing the arguments for color to 2, symbolic name to TG2 and its root info to id=8 (server[2]). No changes to the arguments of the other processes are required.

5.2 From Ensemble ids to MPI ranks

In describing the argument list of processes we have also described most of the processing of routine SetEnvArgs (parsing argv, splitting groups and assigning values to EnvArgs). We have left out the transformation of unique integer process ids, which are Ensemble internal identifiers, to MPI rank identifiers. Ids are absolute unique identifiers known before process spawning; ranks on the other hand are not known before process spawning and are unique within a communicator. Therefore, we must translate ids of ports and roots to their rank within their appropriate communicator. The following code extract of SetEnvArgs constructs two arrays Ranks2Ids and Ids2Ranks, for each new context (field ActualComm of EnvArgs indexed by CurrentContext). Array Ranks2Ids associates ranks to ids of processes in the group (Size is the number of processes in the group) and is constructed by MPI_Allgather. Array Ids2Ranks is obtained by inverting Ranks2Ids and associates Ids to Ranks. It has WS+1 elements, as ids start from 1, where WS is

the number of processes in MPI_COMM_WORLD. Elements of Ids2Ranks corresponding to ids not participating in the group are set to –1.

```
int *Ranks2Ids, *Ids2Ranks, I;

Ranks2Ids=(int*)malloc(Size*sizeof(int));
Ids2Ranks=(int*)malloc((WS+1)*sizeof(int));

MPI_Allgather(&id,1, MPI_INT, Ranks2Ids, 1,
   MPI_INT,
   EnvArgs(CurrentContext).ActualComm);

for (I=0; I<=WS; I++) Ids2Ranks[I]=-1;
for (I=0; I<Size; I++)
        Ids2Ranks[Ranks2Ids[I]]=I;
```

The rank of a port or root is obtained from its id by Ids2Ranks[id] and stored in the appropriate elements of EnvArgs. Upon completion of the SetEnvArg routine, all MPI envelope-related data required in executables is in the appropriate elements of EnvArgs. By calling "mpirun –globusrsl composition-script.rsl" applications are spawned.

6. Conclusions

We presented two fundamental extensions of Ensemble Methodology. The first is that Virtual Component Code supports all communication MPI routines and it is very close to MPI pure code, apart from the call of SetEnvArgs routine and the use of macros instead of envelope related expressions in MPI calls. Pre-processing expands macros and the envelop arguments get their appropriate bindings to EnvArgs elements. The second extension is its implementation on top of MPICH-G2.

Modular components and their co-operation is a fundamental requirement in Grid computing. We have demonstrated Ensemble Grid programming by two designs of a simple problem. The all-terminal design is a regular SPMD application where Ensemble does not offer any significant advantages. The terminal server design however, is a challenging design and demonstrates how modular components used in SPMD (terminals), may co-ordinate with other modular components (servers) in MPMD without any code modifications. Terminals and servers may be grouped in any configuration by appropriate arguments. With current programming techniques each configuration would require source code modification.

Plans for future work are re-designing real SPMD programs as modular components and use them in MPMD Grid applications. Until now we have only experimented with small demonstrating programs, which show the feasibility of complex MPMD compositions, but do not solve real problems.

We will also explore the relation between component generality and modularity. If for example in the terminal code, the two collective calls Reduce and Broadcast were replaced by All_Reduce the code could not be re-used in the second design. The server and terminal design relied on the separation of the two collective calls, so that between the two calls servers compute the max. Generality built within code influences component modularity.

Finally, we plan to extend GrEnT to design parameterised symbolic topologies and to manage underlying grids.

Acknowledgment. This work has been partially supported by the Special Account for Research of the University of Athens.

7. References

1. Cotronis, J.Y. (1996) Efficient Composition and Automatic Initialisation of Arbitrarily Structured PVM Programs, in *Proc. of 1st IFIP International Workshop on Parallel and Distributed Software Engineering*, Berlin, 74-85, Chapman & Hall.
2. Cotronis, J.Y. (1996) Efficient Program Composition on Parix by the Ensemble Methodology, in *Proc. of Euromicro Conference'96*, Prague, IEEE Computer Society Press.
3. Cotronis, J.Y.:(1997) Message Passing Program Development by Ensemble, *Proc. PVM/MPI'97*, LNCS **1332**, 242-249, Springer.
4. Cotronis, J.Y. (1998) Developing Message Passing Applications on MPICH under Ensemble, in *Proc. of PVM/MPI'98*, LNCS **1497**, 145-152, Springer.
5. Foster, I. (1995) *Designing and Building Parallel Programs*, Addison-Wesley Publishing Company, ISBN 0-201-57594-9.
6. Foster, I., Kesselman, C (eds.) *The Grid, Blueprint for the New Computing Infrastructure*, Morgan Kaufmann, 1999.
7. Foster, I., Geisler, J, Gropp, W, Karonis, N., Lusk, E., Thiruvathukal, G., and Tuecke S.: Wide-Area Implementation of the Message Passing Interface, Parallel Computing, 24(12):1735-1749, 1998.
8. Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R. and Sunderam, V. (1994) PVM 3 User's guide and Reference Manual, ORNL/TM--12187.
9. Globus Quick Start Guide, Globus Software version 1.1.3 and 1.1.4, February 2001. www.globus.org
10. Gropp, W. and Lusk, E. (1999) User's Guide for mpich, a Portable Implementation of MPI, ANL/MCS-TM-ANL-96/6 Rev B
11. Message Passing Interface Forum (1994) *MPI: A Message Passing Interface Standard*.
12. MPICH-G2, http://www.hpclab.niu.edu/mpi/g2_body.html