In Proceedings 8th PDP2000 Euromicro Workshop on Parallel and Distributed Processing, IEEE Computer Society press, pp. 398-405

Reusable Message Passing Components

J. Y. Cotronis

Department of Informatics, University of Athens, Panepistimiopolis, 15771 Athens, Greece
cotronis@di.uoa.gr

Abstract

Reusability of executables is an integral part in the design and implementation of message passing programs, as from the same executable component a number of interacting processes are spawned. In practice, reuse of executables within an application is usually achieved when processes execute within a specific regular topology and within a specific message passing environment (e.g. PVM, MPI). We propose the design and implementation of reusable message passing components, which are independent of the target message passing environment (MPE) and may be used in any topology, whether regular, partially regular or irregular. We define virtual process communication interfaces independent of any topology. Process topologies are established by associating compatible communication interfaces.

1. Introduction

In sequential programming reuse of components is considered as using already developed software artefacts in future applications. But in parallel programming reusability is an integral part of the design and implementation of a single parallel program. It is very common and many times desirable for processes to be spawned from the same executable. This is certainly true in message passing programming, where the SPMD model is the extreme case, in which all processes are spawned by the same executable component. The task a designer of a message passing program faces is to express in a single component the appropriate interactions of all the processes, which will be spawned from it, considering all their possible positions in the topology and additionally, for any size of the topology. Then the design has to be implemented using a message passing environment, such as PVM [6] or MPI [9] or Parix [10].

Process topologies in message passing applications are implicitly specified, but explicitly programmed within components. Topologies are formed by considering processes as nodes and communications channels between processes as arcs. Communication channels, i.e. the passing of messages from one process to another, in point-to-point communication are expressed by symmetric calls of send and receive operations in two processes. Each of the send or receive operations in one process has as parameter the identifier of the other process (e.g. tid in PVM and rank in MPI). Other parameters, which are common in both sides, include an integer value tagging the message and an implicit or explicit context. Process topologies are implicitly specified by such pairs of symmetric calls, but must be explicitly programmed by manipulating process identifiers in the calls. If the topology is regular, e.g. a grid or a ring, the designer develops functions, which take a process identifier and return the identifiers of its communicating processes. These functions are usually parameterised to return the identifiers processes in any size of the regular topology.

There are three problems with this approach. The first is that the implementation effort does not only depend on the application design, but also on the target Message Passing Environment (MPE), such as PVM and MPI. Each MPE requires different implementation techniques for programming the same design, because of the process management models they assume. Each MPE is suited for specific types of process topologies, those being closer to its process management model. For example, PVM favours tree and MPI ring or grid topologies. Topologies not well suited to an MPE may certainly be implemented, but require more complex programming. Additional effort is required for parameterizing topologies not well suited to a particular MPE [1, 2, 3, 4].

The second problem is that this approach is only suitable for regular topologies. There are many topologies, which are not globally regular but only partially or locally regular or even altogether irregular. In such cases, general functions returning the identifiers of the communicating processes cannot be derived and consequently ad hoc programming methods are used.

The third problem is that programming communication channels based on symmetric calls of send and receive operations which directly use specific process identifiers limit the reuse of components, as the processes spawned from them may operate only within specific topologies.

In this paper we present the design of reusable message passing components, which are independent of the target

MPE and may be used in any topology, whether regular, partially regular or irregular. We define generic process communication interfaces independent of any topology, which processes use in the parameters of point-to-point and group operations. We establish process topologies by associating compatible communication interfaces.

This paper is structured as follows: In section 2 we present the requirements for generic point-to-point process interfaces and demonstrate their use in constructing regular, partially regular or irregular applications. In section 3 we design open process interfaces for point-to-point, as well as for group communication and barrier synchronisation. In section 4 we implement reusable components. In section 5 we outline the composition of reusable components in the Ensemble methodology [1, 2, 3, 4]. Finally, in section 6 we present our conclusions and plans for future work.

2. Process communication interfaces and process topologies

The physical analogy to the reusable components we try to construct is manufacturing of components, which are assembled or composed to make different products. Another analogy is Lego toy components, in which the same pieces are used to make different constructions. The reason for being able to reuse such components is the design of their interfaces, which permits components to be plugged together. Our objective is to be able to specify such interfaces for processes in a message passing environment. Let us examine the requirements for such process interfaces in point-to-point communication.

2.1 Process interface types for point-to-point communications

We may consider point-to-point communication interfaces as consisting of a number of abstract communication ports, through which messages may be sent and received. We may distinguish two main types of interfaces depending on the way these ports are related and managed.

Fixed number of ports. This is the simplest interface type and has the same number of ports in all processes. Typical example is the interface of processes in a ring topology, where all processes have two such interface types for connecting to the previous and next processes, each having exactly one port (figure 1).

Variable number of ports. An interface type may involve a variable number of ports, which are managed in a semantically similar way. We may distinguish two cases. The first has a variable, but bounded number of ports. The number of ports in this case may be any number within a fixed range from 0 or 1 up to any number N. The interface of a process spawned from a component having such an interface type has a number of ports within the specified range. Typical example is the interface of processes is in a grid topology, where all processes have four such interface types, for North, South, East and West each having either none or one port (figure 1) depending on its position on the grid.

The second case of a variable number of ports has an unbounded number of ports. The number of ports in this case may be any number from 0 or 1 up to any number. Interface processes spawned from a component having such interface types have any number of ports. Typical example is the interface of a master process, which may have any number of slave processes (figure 1).

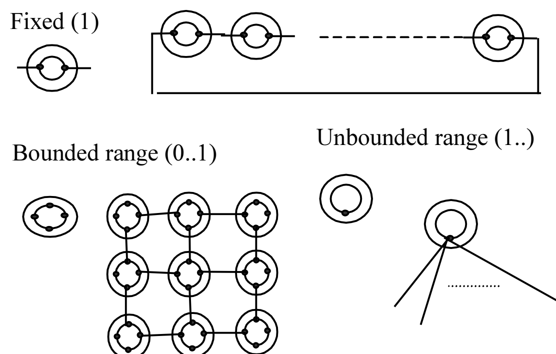


Fig. 1: The three process interface types exemplified in components used in regular topologies

2.2 Examples of partially regular or irregular topologies

The three process interface types have already been used in examples (fig. 1) of regular topologies. Let us now give some examples of processes having the same three interface types and construct locally or partially regular and irregular topologies.

Locally or partially regular topologies. There are topologies, which are not globally regular, but contain nevertheless regular topologies. We demonstrate such a topology by the application Get Maximum. The requirement is simple: Selector processes get an integer parameter and require the maximum of these integers.

We design the following solution, called Selector-Servers-in-Ring: Selectors are connected as client processes to some associated Server processes. Each Selector sends its integer parameter via its single port of interface type Out (type fixed to 1) to its associated Server and, eventually, receives the required maximum via its single port of interface type In (fixed to 1) from the same Server. Servers receive integer values via their Cin ports (unbounded type) from their client Selectors and find the local maximum. Servers are connected in a ring. They find the global maximum by sending via Pout port (fixed to 1) their current maximum to their next neighbour Server in

the ring, receiving via Pin port (fixed to 1) a value from their previous Server neighbour, comparing and selecting the maximum of these two values. Servers repeat the send-receive-select cycle M-1 times, where M is the size of the ring. Finally, Servers send the global maximum via Cout ports (unbounded) to their client Selector processes. The topology or process communication graph (PCG) of the design is depicted in fig.2.

Note that Server processes have ports of two types of interfaces: Pin and Pout are fixed to 1 and are used to construct the ring of Servers, and Cin and Cout of the unbounded variable type, used to construct the client/server topologies with Selectors. The design is partially regular, as each Server is associated with its own Selectors by a regular server/client topology and Servers are associated in a regular ring topology.

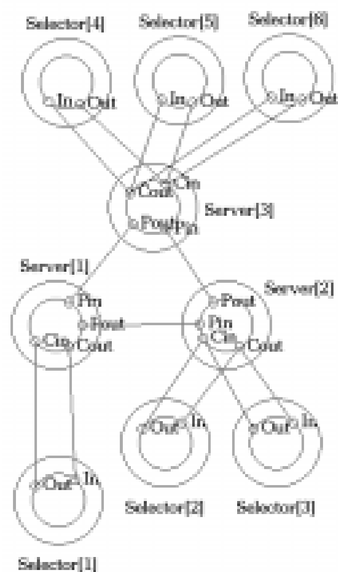
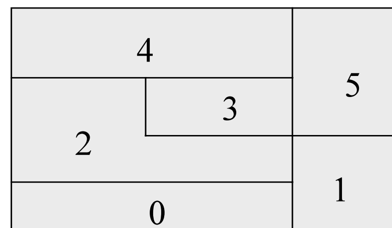


Fig. 2. The Selector and Server in a ring design

We may add to the design more Selectors to any Server by scaling the Server's interface accordingly and setting the appropriate channels. We may also add more Servers in the ring. Implementing reusable components for this type of topology involves the problems we noted in the introduction.

Irregular topologies. There are also topologies, which are completely irregular. For example, consider an implementation of a PDE solution by message passing, but partitioning data into irregular blocks (fig. 3), possibly for load balancing.

The process topology obtained by allocating these irregular blocks to processes is itself irregular (fig. 4). This type of topology is very difficult to implement. Certainly there are not general methodologies for constructing reusable executables in irregular of topologies.



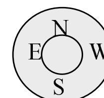


Fig. 3: Irregular partition and component

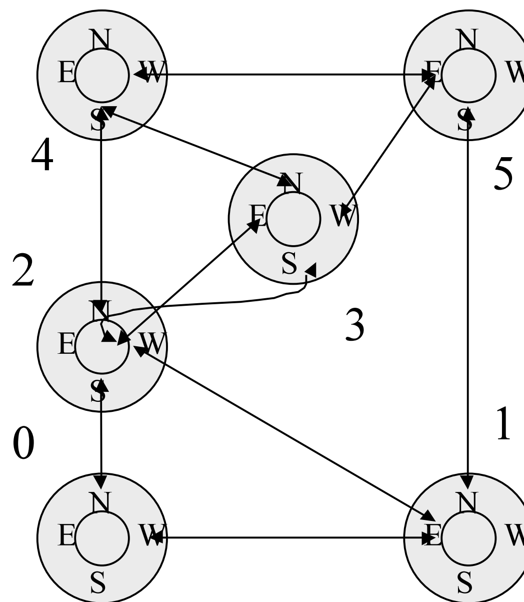


Fig. 4: Irregular topology with six processes

In this section we have demonstrated that, at least at the design level, we may construct process topologies with three generic interface types. In the sequel we will present the implementation of these interfaces and the construction of reusable components.

A topology is then composed in three simple steps: spawn processes, scale their interface if necessary and specify communication channels by setting port information.

3. Implementation of open process interfaces

In this section we implement open process interfaces (OPI) for point-to-point as well as for collective communications. The implementation will be on top of specific MPEs and in spite of their syntactic and semantic differences the OPIs may be applied to any MPE.

3.1 Point-to-point communications

The notion of a port is not explicit in message passing interfaces, but implicit because send and receive operations use process identifiers as parameters identifying the receiver and respectively the sender process. The notion of a communication channel is also conceptual and refers to the passing of a message from one process to another, as determined by the symmetric send/receive calls.

Single port and arrays of similar ports. We first make the notions of port and communication channel explicit. We consider a port as being a structure which holds information of the recipient or the sender process, used respectively by send and receive operations. In PVM3.3 for example, a port would be defined as a structure `struct port={int tid, msgtag;}`, which holds pairs of (tid, msgtag) the parameters denoting message destination in send and respectively, origin in receive operations. A channel between two ports p1 and p2 exists if : they belong to processes tid1 and tid2 respectively, they have a common msgtag, and the tid of p1 is tid2 and the tid of p2 is tid1 (i.e. p1.msgtag=p2.msgtag ^ p1.tid=tid2 ^ p2.tid=tid1). A port in PVM3.4 as well as in MPI would have an extra field for the context of the message (communicator in MPI) and a channel between two ports would additionally require that the ports have a common context.

Ports of the same type (such as the Cin and Cout ports of Server) form arrays, the number of which is a variable in general and has to be dynamically allocated. We define the structure

```
struct port_type
       {int portcount; port_struct *port;}
```

which is a header element with two fields, one for holding the actual number of ports (portcount) and a pointer to the actual array of ports of the process to which it belongs. Note that the above declaration is independent of any specific message passing interface. The differences are hidden in the declaration of a single port.

Structure of Ports and its use in point-to-point communication routines. Finally we define the complete interface as an array of port_types as

```
struct *port_type Interface;
```

Such port_types are associated with the interface types of components, e.g. Pin, Pout, Cin, Cout of Server and In, Out of Selector.

In the body of the components all send and receive operations in processes refer to ports, identified by the appropriate port type and the port index within the type. In PVM for example, tid and msgtag parameters of pvm_send and pvm_recv routines should refer to tid and msgtag of a port P of type T, i.e. Interface[T].port[P].tid and Interface[T].port[P].msgtag, respectively. Alternatively, we may hide pvm_send and pvm_recv calls within wrapper routines, i.e. `Wsend(T,P)`, `Wrecv(T,P)`, where T is an index to an element of Interface and P and index of a port within this element. With this convention sending to the i'th port of Cout may be written as `Wsend(Cout,i)`.

We have achieved developing message passing code without referring explicitly to processes, but to ports. Ports are virtual at compile time as they do not have any values. Also their actual number in each port type may be unspecified at compile time. This is what makes process interfaces open and scalable and the code that uses such interfaces reusable. In section 4 we explain how the number of ports and their values are set at run time.

3.2 Collective communications

We may apply the idea of defining point-to-point communication parameters in a virtual way to collective communications, namely barriers and groups. In the way we have given virtual names, namely port types, for specifying the parameters of point-to-point communication, we may also give virtual names to barriers and groups.

Single Barrier or Group and its use in collective communication. For a virtual barrier in PVM we only need the simple structure

```
struct barrier_type {char *group;}
```

and for a group the structure

```
struct group_type {char *group; int msgtag}
```

Each process spawned from the component will be able to participate in a barrier or group the actual name of which will be stored in `*group`. This way we distinguish between the fact that every process spawned from the same component should participate in a barrier or group, but each process may in principle participate in a different barrier or group. The name of the actual barrier or group and the msgtag must be provided dynamically at run time time. Consider the case of an SPMD application where the processes are spawned from the same component form a tree topology and we require that processes on each level of the tree use the same barrier. The component specifies that each process spawned from it participates in a barrier, but not any specific barrier and certainly not the same barrier. In principle, each process may participate in a different barrier.

3.3 General Interface Structure

The complete interface structure is declared as

```
struct *g_type Interface;
```

where `g_type` is a union structure of the three cases above

```
union g_type
  {  struct port_type;
     struct barrier_type;
     struct group_type;}
```

All barrier and group parameters of collective routine calls in the body of the components will refer to the corresponding element of the general interface structure. E.g. a PVM broadcast call will be

```
Info=pvm_bcast( Interface[T].group,
                Interface[T].msgtag)
```

and a barrier call will be

```
Info=pvm_barrier(Interface[T].group, count)
```

4. Implementation of Reusable Components

Reusable message passing components should only compute results and do not involve any process management or assume any topology in which they operate. If message passing routines in the code use the virtual interface ports, barriers and groups we constructed in the previous section they already have this property.

Of course we must pass information to the interface of each process upon its creation to establish the required process topology. We assume at this stage that the process topology is somehow specified externally. Many design tools such as Trapper [12], Grade [7,8], EDPEPPS [5], etc. provide features for specifying process topologies. We assume also that there is a "loader" program which interprets such topology designs and establishes the actual process topology in three steps: spawns processes in the topology, provides the actual number of ports of each port type of each process and finally provides values to ports, groups and barriers.

The aforementioned design tools do not make such clear distinction between the process as it looks from the outside (used to specify the process topology) and the code of the processes from inside. In other words they do not use reusable components. In section 5 we describe the Ensemble methodology, which makes this distinction clear, makes use of the reusable components and may be integrated in any of the design tools mentioned before.

4.1 Implementation of open interfaces

Looking components from inside, we need to implement two aspects to make them reusable: setting the number of ports of each interface type and set values for the fields of ports, barriers and groups.

Setting number of ports. As processes are spawned, the actual number of ports for each port type may be passed by the loader programme as command line parameters. The first procedure processes call is MakePorts, which reads these parameters and allocates space for the appropriate number of ports and sets the portcount field to the appropriate value for each port type in array Interface.

Setting Interface with communication parameters. Processes then get the values for the communication parameters for each port. Processes set actual values to their interface by executing a routine, called SetInterface. Each MPE requires its own implementation of the SetInterface routine. In MPI, as process identification (rank) is known at compile time, it is possible to pass it in their command line parameters. In PVM, where process identifiers tid are dynamically determined, it is not possible in general to pass to a process at the time of its spawning the tids of its neighbours, as these may have not been spawned yet. In PVM the "loader" program sends messages with the tids of the neighbouring processes as data. Symmetrically, the SetInterface receives these messages and updates the interface.

The Main Body of Reusable Components. Having constructed and set values to the interface the components may proceed with the application computations. A common Main for all program components, which may be seen below has been developed. The application computations for each component are coded in RealMain routine.

```
/* Common Main for all reusable components */
void main(argc,argv) int argc; char **argc;
{  MakePorts(Interface);
   SetInterface(Interface);
   RealMain(Interface,argc,argv);}
```

The programmer has only to write the RealMain routine.

4.2 Code parameterisation

Sometimes the number of messages sent or received depends on the size of the topology (local or global) a process belongs to. For example, for the servers of fig. 2 to operate correctly they must repeat the send-receive-select cycle M-1 times, where M is the size of the ring they belong to. These parameters are not application parameters, in the sense that they are not demanded by the requirements of the problem. They are introduced from the particular design of the solution, and for this reason are called design parameters. Since design parameters depend on the discrete topology size are always non-negative integers and may be passed as command line parameters to the spawned processes, together with any application parameters. The first action of `RealMain` routine is to read the design and application parameters.

4.3 Reusable components

We have now completed all the elements of the reusable components. The `RealMain` of the Selector and Server programs are outlined in fig. 5.

For convenience, we have abstracted PVM send and receive routines by using `send(T,P,What)` and respectively `receive(T,P,What)`, to indicate that message What is sent to, and respectively, received from port P of type T. The same wrapper routines may call MPI routines if the target MPE is MPI, thus hiding all differences and maintaining one source code of components for any MPE.

5. Composing message passing applications

We have used such reusable components in the Ensemble methodology and its tools for the design and implementation of message passing programs independent of any message passing environment. Ensemble specifies a common software architecture, which is depicted in figure 6, for all message passing programs on any MPE.

An application in Ensemble is an "ensemble" of reusable executable program components and of a script which specifies the application processes (to be spawned from the reusable program components), their topology (or Process Communication Graph-PCG) and the mapping of processes onto the virtual architecture. The programmer specifies the PCG and encodes the source programs (using abstract communication routines as in 4.3) independent of the target architecture and MPE. By compiling and linking with the appropriate libraries the reusable executables (within the specific architecture and MPE) are obtained. The programmer also specifies the resource allocation (mapping of processes to processors, data files, etc.)

Universal (within an MPE) tools, PCG-Builder, Annotator and the Loader program, interpret the script and compose the message passing application by spawning application processes and establishing their communication channels relieving the programmer of a complex task.

Ensemble has been applied [1,2,3,4] to compose applications in PVM, MPI and Parix, three very different MPEs; each requiring its own techniques for reusable components and its own loader program. Using the Ensemble methodology applications may be ported without much difficulty to other message passing environments and in certain cases automatically. Process Topologies or PCGs are either represented by the Ensemble script language or directly by GrEnT, a simple Graphical Ensemble Tool, demonstrating that it is possible to integrate the Ensemble methodology in existing graphical environments supporting the design, implementation and debugging of message passing programs.

The Ensemble script describing the topology of the design of Get maximum application is shown in fig. 7. The first section of the script headed by Components specifies the components to be used and their interface. The second section headed by Processes specifies the processes with the appropriate number of ports and values for their design parameters. The third section headed by Channels specifies port associations. The application parameters are specified under a separate section.

To demonstrate the reusability of the components we give another design to the same problem (fig. 8). We modify the topology into a tree. Server[1] and Server[2] have the same clients as in the previous design. Server[3] now has Selector[6] as its client but also Server[1] and Server[2], connected through their Pin and Pout ports. Server[3] has no connections to its Pin and Pout ports. Sever[1] and Server[2] receive values from their clients and find their local maximum; they send their local maxima to their Pout port, which is connected to a Cin port of Server[3]. Server[3] finds the global maximum, and as it is not connected in a ring, it does not perform any send-receive-select cycle over its Pin and Pout ports; it sends the global maximum to its clients. The two Server clients receive and select the global maximum and send it to their Selector clients.

Further information on Ensemble may be found on the Web site http://www.di.uoa.gr/~ensemble.

```
        /* Selector Code */
#define In    0
#define Out   1



void RealMain(Interface,argc,argv)
   struct port_types *Interface;
   int argc; char **argv;
{  GetParams(N);
   send(Out,1,N);
   receive(In,1, Max);
}
```

```
                    /* Server Code */
#define Pin  0
#define Pout 1
#define Cin  2
#define Cout 3

void RealMain(Interface,argc,argv)
   struct port_types *Interface;
   int argc; char **argv;
{  GetParam(M);
   LMax=0;
   for (j=1;j<=Interface[Cin].PortCount;j++)
     {receive(Cin,j,V); if (V>LMax) Lmax=V; }
   Gmax=Lmax;
   for (i=1;i<=M-1;i++)
   { send(Pout,1,GMax);
     receive(Pin,1,V);
     if (V>GMax) Gmax=V}
   for (j=1;j<=Interface[Cin].PortCount;j++)
   Send(Cout,j,Gmax)}
```

Fig. 5: The code of selector and server

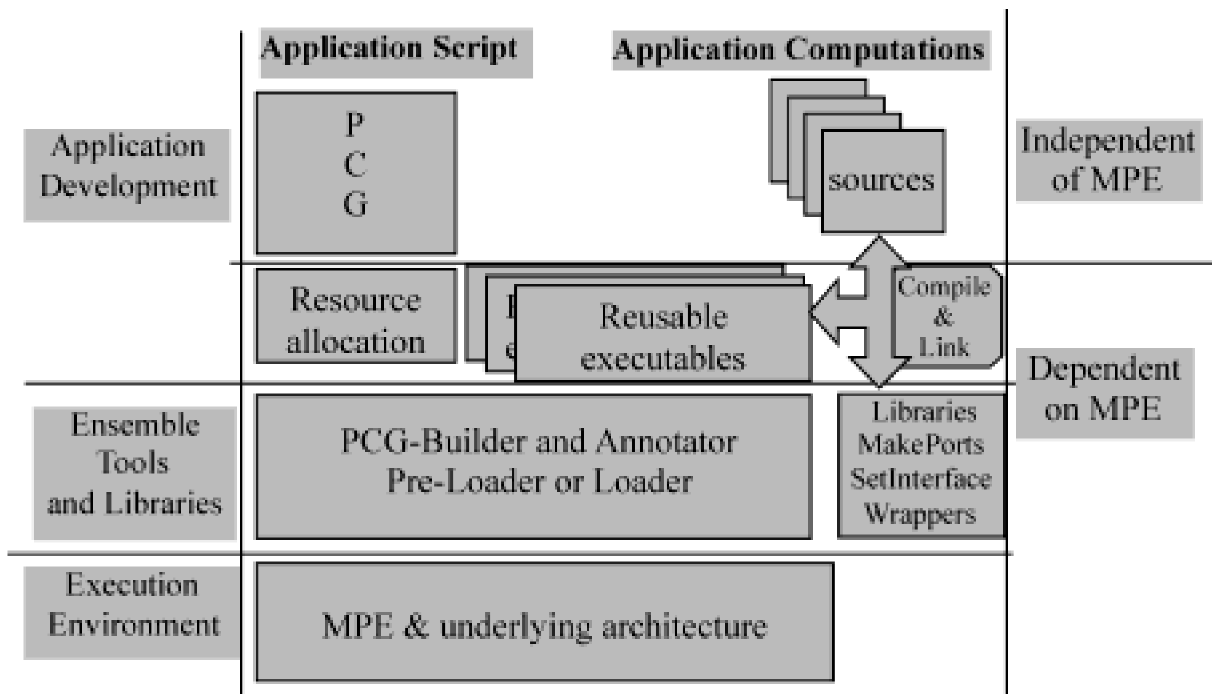


Fig. 6: The Ensemble software architecture

APPLICATION Get_Max_Selector_Servers_in_Ring;

PCG
Components
Selector **range**: In, Out [1..1]
Server **range**: Pin, Pout [0..1], Cin, Cout [0..]; **DParams** M;
Processes
Selector[1], Selector[2], Selector[3],
Selector[4], Selector[5], Selector[6] **#ports** = In:1, Out:1;
Server[1] **#ports** = Cout, Cin:1, Pout, Pin:1; **DParams** M=3;
Server[2] **#ports** = Cout, Cin:2, Pout, Pin:1; **DParams** M=3;
Server[3] **#ports** = Cout, Cin:3, Pout, Pin:1; **DParams** M=3;
Channels
Selector[1].Out[1] -> Server[1].Cin[1]; Server[1].Cout[1] -> Selector[1].In[1];
Selector[2].Out[1] -> Server[2].Cin[1]; Server[2].Cout[1] -> Selector[2].In[1];
Selector[3].Out[1] -> Server[2].Cin[2]; Server[2].Cout[2] -> Selector[3].In[1];
Selector[4].Out[1] -> Server[3].Cin[1]; Server[3].Cout[1] -> Selector[4].In[1];
Selector[5].Out[1] -> Server[3].Cin[2]; Server[3].Cout[2] -> Selector[5].In[1];
Selector[6].Out[1] -> Server[3].Cin[3]; Server[3].Cout[3] -> Selector[6].In[1];

Server[1].Pout[1] -> Server[2].Pin[1];
Server[2].Pout[1] -> Server[3].Pin[1];
Server[3].Pout[1] -> Server[1].Pin[1];

APPLICATION PARAMETERS
Selector[1]: "6"; Selector[2]:"999"; Selector[3]:"7";
Selector[4]:"8"; Selector[5]:"9"; Selector[6]:"5";

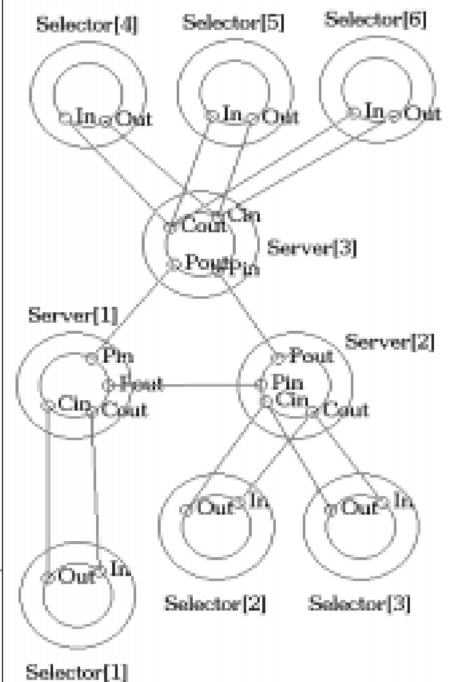
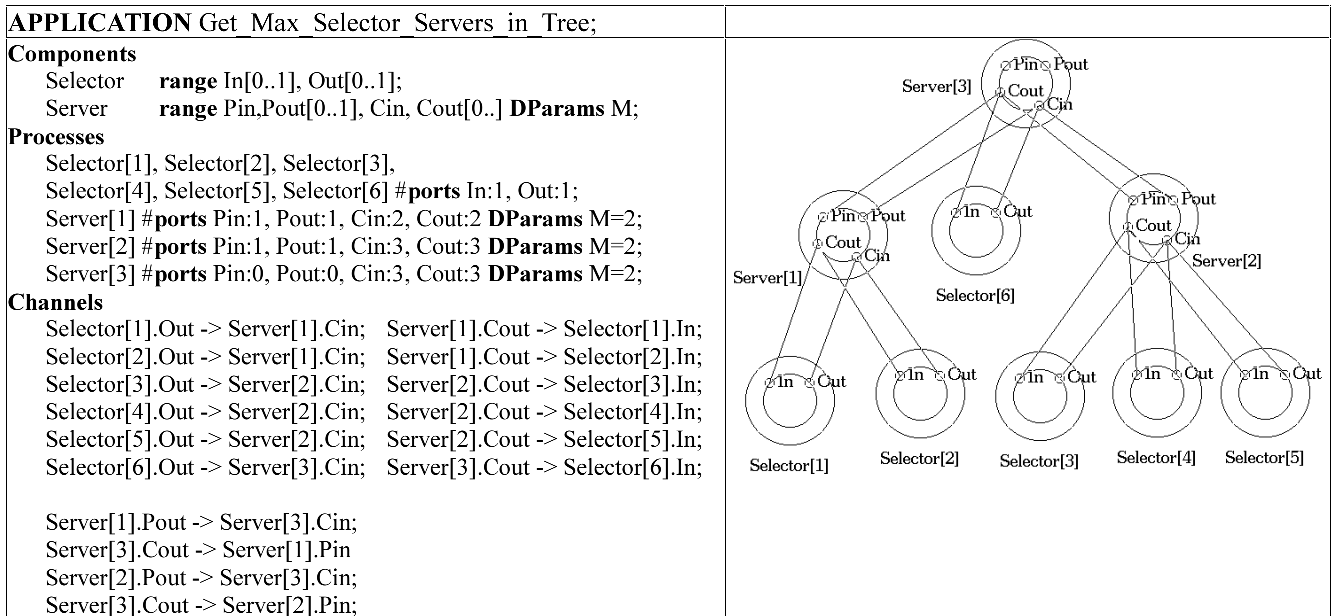


Fig. 7: The Ensemble script defining topology for the Selectors and Server in a Ring



```
APPLICATION Get_Max_Selector_Servers_in_Tree;
Components
    Selector    range In[0..1], Out[0..1];
    Server      range Pin,Pout[0..1], Cin, Cout[0..] DParams M;
Processes
    Selector[1], Selector[2], Selector[3],
    Selector[4], Selector[5], Selector[6] #ports In:1, Out:1;
    Server[1] #ports Pin:1, Pout:1, Cin:2, Cout:2 DParams M=2;
    Server[2] #ports Pin:1, Pout:1, Cin:3, Cout:3 DParams M=2;
    Server[3] #ports Pin:0, Pout:0, Cin:3, Cout:3 DParams M=2;
Channels
    Selector[1].Out -> Server[1].Cin;   Server[1].Cout -> Selector[1].In;
    Selector[2].Out -> Server[1].Cin;   Server[1].Cout -> Selector[2].In;
    Selector[3].Out -> Server[2].Cin;   Server[2].Cout -> Selector[3].In;
    Selector[4].Out -> Server[2].Cin;   Server[2].Cout -> Selector[4].In;
    Selector[5].Out -> Server[2].Cin;   Server[2].Cout -> Selector[5].In;
    Selector[6].Out -> Server[3].Cin;   Server[3].Cout -> Selector[6].In;

    Server[1].Pout -> Server[3].Cin;
    Server[3].Cout -> Server[1].Pin
    Server[2].Pout -> Server[3].Cin;
    Server[3].Cout -> Server[2].Pin;
```


Fig. 8: The Ensemble script defining topology for Selectors and Server in a Tree

6. Conclusions

We have designed and implemented reusable message passing components, which do not rely on any specific topology. The reusable components have open and scalable interfaces, which may be used to compose and scale any topology, be it regular, irregular or locally regular.

We have demonstrated their reusability within the Ensemble methodology by implementing two designs of the Get Maximum problem, which reuse the Selector and Server components; it was sufficient to change only the topology of the application.

The reusable components and the Ensemble architecture may be integrated into any tools supporting the design, implementation and debugging of message passing applications.

7. References

[1] Cotronis, J.Y.: Efficient composition and automatic initialization of arbitrary structured PVM programs. IFIP Proceedings of the 1st Workshop on Software Engineering for Parallel and Distributed Systems, (held in association with ICSE 96), Berlin, March 1996.

[2] Cotronis, J.Y.:Efficient Program Composition on Parix by the Ensemble Methodology. In: Proc. 22nd Euromicro Conference 96, Computer Society Press, 1996.

[3] Cotronis, J.Y.: Message--Passing Program Development by Ensemble.In: Bubak, M., Dongarra, J. (eds.): Recent Advances in PVM and MPI. LNCS Vol. 1332, Springer--Verlag, Berlin Heidelberg New York (1997) 242—249

[4] Cotronis, J.Y.: Developing Message--Passing Applications on MPICH under Ensemble.In: Alexandrov, V., Dongarra, J. (eds.): Recent Advances in PVM and MPI. LNCS Vol. 1497, Springer--Verlag, Berlin Heidelberg New York (1998) 145—152

[5] Delaitre T., Zemerly MJ, Justo G R, Spies F and Winter S (1998): EDPEPPS: An Environment for Optimal Parallel Software Design,*Journal of Computers and Artificial Intelligence,* 17(5), pp. 405-416, 1998.

[6] Geist, A. , Beguelin, A. , Dongarra, J. , Jiang, W. , Manchek, R. , Sunderam, V.:PVM 3 User's guide and Reference Manual. ORNL/TM--12187, May 1994

[7] Kacsuk, P., Dózsa, G. and Fadgyas, T. (1996) Designing Parallel Programs by the Graphical Language GRAPNEL, *Microprocessing and Microprogramming*, 41, 625-643.

[8] Kacsuk, P., Dózsa, G., Fadgyas, T. and Lovas, R. (1998) The GRED Graphical Editor for the GRADE Parallel Program Development Environment, in *Proc. of HPCN98, International Conference on High-Performance Computing and Networking,* 728-737.

[9] Message Passing Interface Forum (1994) MPI: A Message Passing Interface Standard.

[10] Parsytec Computer Gmbh.:Report Parix 1.2 and 1.9 Software Documentation.1993

[11] Scheidler, C. and Schafers, L. (1993) Trapper: A graphical programming environment for industrial high-performance applications, *in Proc. of PARLE'93: Parallel Architectures and Languages Europe*, Munich, Germany.