A Concurrent Programming Support for Distributed Systems

G. Spezzano and D. Talia CRAI, Italy
M. Vanneschi University of Pisa, Italy

ABSTRACT: This paper describes a concurrent programming support implemented on a distributed architecture. The concurrent programming model is derived from the Communicating Sequential Processes (CSP), with some extensions to allow asymmetrical and asynchronous communications; furthermore, some statements for fault handling have been defined.

The system described here, is named *NERECO* (NEtwork REmote COmmunications). It is composed of a concurrent language and a set of static tools and a run-time support for the design and the implementation of concurrent distributed applications on a network of computers. The NERECO system has been implemented in C on a network of Sun workstations.

1. Introduction

The design and implementation of efficient and reliable distributed applications require high-level tools and mechanisms characterized by expressive power, high modularity, and robustness. Although the operating system and the low levels of interprocess communication offer mechanisms for process creation and cooperation, the development of the greatest part of distributed applications requires a set of constructs that are more powerful and at a higher level [Bal et al. 1989; Hansen 1973].

A high-level distributed concurrent programming support offers an abstraction level in which resources are defined like abstract data types encapsulated into processes. According to this approach, a distributed program consists of a set of processes cooperating by message passing and located on one or many computers. In the implementation of typical distributed programming techniques (e.g., concurrent activities management, data and processes replication, synchronization, and fault tolerance), a concurrent programming support isolates the distributed software designer from the underlying network architecture, communication protocols and operating system.

The aim of this concurrent programming support is to provide a methodology for modular and robust structuring of distributed programs by:

- characterizing the processes in a functional way and associating a type to each one;
- using unidirectional typed channels;
- expressing communication forms either by point to point (*rendez-vous*) or by diffusion (*broadcast* and *multicast*);

- controlling nondeterminism in communications; and
- handling, in a simple and flexible way, fault conditions through detection, confinement and recovery.

The system described here is the first of a set of tools which has been developed at CRAI (Consorzio per la Ricerca e le Applicazioni di Informatica) in order to support the development of efficient and reliable distributed software. It is the prototype of a distributed support system for the development of concurrent distributed programs termed NEtwork REmote COmmunications (NERECO) [Spezzano et al. 1987; DeFerrani et al. 1985].

This paper is organized as follows. Section 2 gives an overview of the system. Section 3 describes the language constructs. In section 4 the static tools are presented. Section 5 shows one example program written in the language. Finally, in section 6 the distributed run-time support of NERECO is described.

2. *NERECO Overview*

This section gives an overview of the NERECO system with special attention to the design criterions and choices. Several aspects, particularly language constructs and static tools, will be described in the two next sections.

The main goals of NERECO are:

- providing a flexible environment so that users are not bound to one specific set of statements and data types of the sequential part of the language;
- making available few language constructs but that are sufficiently general and powerful to achieve concurrency management, communication facilities and error recovery at a user level;
- making the distributed run-time support for concurrent constructs simple and efficient as much as possible, so allowing extensions for new mechanisms and fault-tolerant requirements;
- guaranteeing a good portability without being bound to a particular host system.

The main issue is to choose the programming language to be offered to the users for developing distributed programs. The first requirement is obviously incompatible with the choice of one of the concurrent languages available at present, such as Ada [Ada 1983], NIL [Strom & Yemini 1983] and CSP-based languages [Hoare 1978] like ECSP [Baiardi et al. 1984; Baiardi et al. 1984a], CSP80 [Jazayeri et al. 1980], Occam [Inmos 1984], Planet [Crookes & Elder 1984], Joyce [Hansen 1987], etc. Actually, the use of a language with powerful abstraction mechanisms for data and control flow, like Ada, could be suitable, but it provides an environment that might be opposed to the needs of many users who want to use a programming style of centralized systems because of system requirements or personal preference. Furthermore, more complex languages contradict third requirement because the complexity of the sequential part has remarkable impact on the concurrent run-time support.

The design choice of NERECO is to add to a sequential language a set of concurrent constructs with well-formed syntax and semantics, and to extend its static development tools with those that support the concurrent part. This approach has been used successfully in other projects, such as Conic [Magee et al. 1986].

Regarding the second requirement, our choice has identified a set of mechanisms derived from the Communicating Sequential Processes (CSP) model. The main features of this concurrent programming model are:

- communication management by means of input and output commands and the use of channels,
- the exploitation of parallelism by means of the parallel command, and
- nondeterminism management by guarded commands.

The CSP characteristics of flexibility and generality which have already been fully tested, have convinced us to choose this model. With respect to CSP, some mechanisms such as asynchronous and broadcast communication, dynamic channels, fault-tolerance statements, and explicit termination, have been added.

Broadcasting is an inexpensive way of communication with a large number of processes. A message broadcast by a process is

received directly by all the other processes in the network instead of being restricted to only one process [Gehani 1984]. Motivated by the characteristics of local area networks, such as Ethernet, the broadcast facility can be used to advantage in designing some kinds of distributed applications.

Dynamic channels allow to change at run time the interconnection among the processes. In this way, a distributed application can be dynamically reconfigured according to the user requirements. Furthermore, dynamic channels can save the amount of channels in a program. For example, when a process must input a value from n processes, *one* dynamic channel can substitute *n* static channels. Finally, used together with fault-tolerance statements, dynamic channels provide the mechanisms to reconfigure the program when a fault occurs.

Only one CSP mechanism has not been considered, the process nesting. This is due to the process granularity of the system on which NERECO has been implemented, that is the heavy weight UNIX processes. The grain size of the UNIX processes suggested us to avoid applications with a very large number of processes that can introduce an excessive overhead decreasing the applications performance.

At the moment NERECO is based on CHILL, Pascal and C languages. The static tools perform:

- the compilation of concurrent constructs,
- the generation of code,
- the control of consistency at process interfaces,
- the location of executable code and configuration files on the network nodes.

The dynamic tools perform:

- the installation of application's processes,
- the distributed run-time support of the concurrent part by processes which interpret concurrent constructs and by processes communicating on the network,
- the load balancing of the distributed application (if the user wish),

- the logging of concurrent construct.

The distributed run-time support of the concurrent constructs has also been implemented by cooperating processes, as a virtual machine on an existing operating system (OS). In this case the OS is Sun UNIX 4.2. The main aspect is that the designer is able to transform typical mechanisms of concurrent languages into system calls easily by a good knowledge of concurrent programming methodologies. The virtual machine has been initially described in a CSP-like language and then "translated" into the C language [Kernighan & Ritchie 1978] with additional UNIX system calls [Ritchie & Thompson 1978], with a limited design effort.

Another advantage of the chosen approach consists of a higher possibility to conceive reconfiguration and fault-tolerant mechanisms in the run-time support implementation. This aspect is essential to isolate the user from problems that can raise by network physical configuration or network reliability.

3. *The language*

As mentioned above, the cooperation model has some differences in comparison with the CSP model, both as extensions and limitations. The major limitation is the lack of a parallel command for the process nesting. In the NERECO system a concurrent program is constituted of a set of processes that are all at the same level and activated at the same instant.

More important extensions are:

- the explicit declaration of message type, to allow complete static type checking and process interface checking;
- the addition of asynchronous and asymmetric communication forms, *multicast* and *broadcast*, which permit to explore a new way of programming distributed applications on local area networks; and
- the use of fault-tolerance constructs to handle communication or process failures.

3.1 Processes

The information which characterizes a process in a distributed program is its *name* and its *type*. The process type is necessary to identify a class of processes, such as *monitor, file server*, etc. This characterization is useful when, for instance, a process needs to operate on a replicated resource available on the network. The process sends the request to all of the resource managers without mentioning the name of each process, but only their type. Inside each process, as its first declaration statement, there is the declaration of the process itself, as follows:

self <*process_id*> : <*process_type_id*> ;

After this declaration, the partners and their type must be declared:

partners <*process_id*>,...,<*process_id*> : <*process_type_id*> ;

3.2 Channels

Channels are "logic objects" realizing the communication among the processes of the program. Like the CSP model, the processes cooperate through communication channels using input/output commands. Channels are typed and identified by the triple:

(sender process set, receiver process, message type)

They can be symmetrical or asymmetrical, and generally they are asynchronous.

A communication channel, always unidirectional, is considered a private object of the single receiver process. It can be static or dynamic; in the first case the name of the partner is represented by a constant, in the second case by a *processname* variable.

The channel message type is composed of a pair (*co, T*), where *co* is the type constructor and *T* is the type offered by the sequential language. In pure synchronization channels, only the constructor is used. Static channels can be defined as follows:

- symmetric and synchronous,
- symmetric and asynchronous,

- asymmetric and synchronous.

Let us show, for example, the syntax of an asymmetric synchronous static channel:

chan from (<*process_id*>,...,<*process_id*>)
type <*constr_id*> (<*msg_type*>);

The <*constr_id*> is the type constructor and with <*msg_type*> constitute the type of message transmitted on the channel. Notice that explicitly declaring the message type it makes possible to check automatically the process interfaces, increasing reliability and making easier the integration test. The processes define, by channels, visible points through which it is possible to make requests and to receive messages. Program security is considerably enhanced guaranteeing that a process can send or receive a message on a channel if and only if the message type is equal to the channel type.

Asynchronous channels are an important feature of the language. They increase the parallelism of the applications, because they avoid the requirement that a sender process waits until the receiver collects the message. To declare an asynchronous channel, it is necessary to specify the *length* of its associated buffer, while the synchronous channels have no length declaration. When the buffer is full, the channel behaves as a synchronous one. Figure 1 shows an example of an asynchronous channel. The *server* process is the receiver, the *user* process is the sender, *update(integer)* is the type of the message, and the channel buffer holds three positions.

```
Process server ::
        .....
  chan from user
        type update (integer);
        length = 3 ;
        .....
```

Figure 1: An asynchronous channel

As mentioned before, channels can be dynamic. In this case the name of the partner process is a variable of *processname* type. Dynamic channels can be symmetric and synchronous.

To allow dynamic channel management it is necessary to define variables of *processname* type, *i.e.* variables whose values are process names. The declaration is as follows:

procvar *<procvar_id>, ..., <procvar_id> ;*

Moreover, it is possible to specify a range in which the values can vary. If the range is not specified the domain of a *processname* variable is constituted of identifiers of all visible processes (self and partners) plus the undefined value. To operate on dynamic channels two constructs are defined:

connect(*<procvar_id>, <process_id>*)

to assign a value and the communication rights, and

detach(*<procvar_id>*)

to assign the undefined value and to revoke whatever communication right. Note that in the communication constructs, channels are not mentioned, but only the names of the partners quoted in the channel declaration, differently from other languages (e.g. Occam), in which channel names are used.

3.3 Communication and nondeterminism

The communications are realized by the I/O commands, **send** and **receive**. The **send** construct can have a symmetrical or asymmetrical form:

send *(<process_id>, <constr_id> (<msg_var>));*

send (all of *(<process_id_list>), <constr_id> (<msg_var>));*

send (all of type = *<process_type_id>,*
<constr_id> (<msg_var>));

In the first case, only a partner exists, it is identified by *<process_id>*; in the second there is a set of partners, defined by a list (*send multicast*); finally in the third the set of partners is defined by a process type identifier (*send broadcast*).

Notice that in the last form of the **send** statement, the user does not specify the list or the number of processes to which the message will be delivered. The *<constr_id>* is the type

constructor defined into the declaration of channel used to send the message.

The syntax of the **receive** statement is:

receive(*<process_id>*,*<constr_id>*(*<msg_var>*));

receive (*<procvar>* : **any of** (*<proc_id_list>*),
<constr_id>(*<msg_var>*));

In the first form, there is only one sender; in the second there is a set of senders, but only one of them delivers the message.

The nondeterministic constructs are similar to those provided by CSP, namely *repetitive* and *alternative* commands with input guards and priority. The syntax of the *repetitive* command is shown below (Figure 2). In the syntax of *alternative* command, the keywords **rep** and **endrep** are replaced by **alt** and **endalt**. The symbol '**%**' separates two branches of the command.

The *<command list>* of the **alt** or **rep** command can be executed only if the *<input guard>* and the *<boolean guard>* have not failed. When all branches are evaluated, only one with a successfully executable guard is selected and executed.

An **alt** command specifies the execution of only one of its branches. A **rep** command specifies as many iterations as possible of its similar **alt** command. When all guards fail, the **rep**

```
rep
  <priority> ;
  <boolean guard> ;
  <input guard> ;
  docl
    <command list>
  endcl ;
%
.............
%
  <priority> ;
  <boolean guard> ;
  <input guard> ;
  docl
    <command list>
  endcl ;
endrep ;
```

Figure 2: The repetitive command

command terminates. Finally, notice that by adding priority it is possible to force a particular scheduling.

3.4 Fault tolerance

The language offers fault handling mechanisms to handle communication failures caused by a:

- physical communication media fault,
- partner termination, or
- channel disconnection.

Failures can be handled by the **onfail, onterm** and **onprot** clauses. These statements can be associated to an input/output command (i.e., send or receive). They make it possible to perform recovery actions (*forward recovery*) when a failure occurs [Spezzano & Talia 1989]. In the example of Figure 3, we used the **onterm** clause as a mechanism to continue the process execution when a communication fails because the partner *server1* is terminated.

```
connect (x, server1);
send (x, exec(param))
onterm
   connect (x, server2);
   send (x, exec(param));
end;
   receive (x, result);
```

Figure 3: Example of the use of *onterm* clause.

3.5 Termination

In the language is defined the explicit termination of a process. A process can terminate at any time by executing the **terminate** construct which lets the process execute termination, informing all the partners. Notice that a process can exclusively execute its termination, but constructs are not provided to force the termination of other processes.

4. Tools

The NERECO system provides a set of static tools to support the development of a distributed program. They are the *precompiler*, the *consistency checker*, and the *configurator*.

4.1 Precompiler

The precompiler essentially carries out the role of a compiler for the concurrent part of the language. It operates on each single process composing the program. Starting from a concurrent program it produces a sequential one. The precompiler output will be the input for the compiler of the sequential language. Notice that it is a "rational preprocessor" [Aho et al. 1986], namely it is not a simple macro translator, but a precompiler for a language enhanced with new data and control structures.

As mentioned before, the dependency of NERECO from a particular sequential language is confined on the precompiler. Hence it is easy to deduce that the choice of a different sequential language in which to embed the concurrent part, involves to change only this module.

In the precompiler design, a lot of attention has been paid to preserve the syntactic and semantic coherence of the host sequential language. After the syntactic and semantic analysis of the concurrent part, the preprocessor generates the sequential code. In the code generation the concurrent constructs are translated into function calls, which contain the code to communicate with the run-time support processes, using the InterProcess Communication (IPC) of UNIX 4.2BSD [Leffler et al. 1983]. Furthermore, the precompiler produces a table containing the necessary information to execute the consistency checks among the processes of the program. Currently, three precompilers are implemented for CHILL, Pascal and C language.

4.2 Consistency checker

The consistency checker performs the static analysis of consistency among the concurrent "objects" of the processes composing the program. In this phase the entity distributed program is created, assigning a name to the set of processes. The consistency checker analyzes the data which have a global interaction on the program, *process names* and *channels.*

The consistency checker analyzes the tables generated by the preprocessor for each process and works out the following operations:

- consistency analysis among the declarations of processes;
- production of diagnostic messages on the insubstantialities;
- synthesis in a global table of the information about all the concurrent objects of the program.

For instance, if process P_1 declares process P_2 as a partner, the consistency checker tests if also the process P_2 declares P_1 as its partner. If not, an error is signaled. The same occurs for a communication channel between two processes.

4.3 Configurator

The configurator provides for the physical configuration of the distributed program on the network nodes. It asks to the user the host name on which each process must run, hence it takes care to transmit the executable files on the corresponding hosts. Finally, it creates a table of correspondence between processes and hosts (*configuration table*) which is useful to the run-time support.

Notice that having a distributed file system, like that of Sun UNIX 4.2 [Lyon et al. 1984], it is not necessary to allocate the executable files on the network hosts. In this case the run-time support loads on each node the executable code from the distributed file system according to the configuration table.

Once developed, a program can be configured in all the possible ways without changing the code of the processes. This is possible because the concurrent constructs are independent from the particular process location. The distributed run-time support provides the message routing on the basis of the configuration table.

Furthermore, we enhanced the configuration facilities by putting in the run-time support a tool for a load-balancing configuration that users can optionally utilize.

5. *An example*

This section presents an example of concurrent programming in the NERECO language. This program is a simple example of a distributed implementation of partitioned or replicated resource management. In the program a process manages a set of resources, in this case a set of counters. There are three types of processes, *Allocator, Counter* and *User.* The *Users* can request to the *Allocator* process the services of the *Counter* processes. After

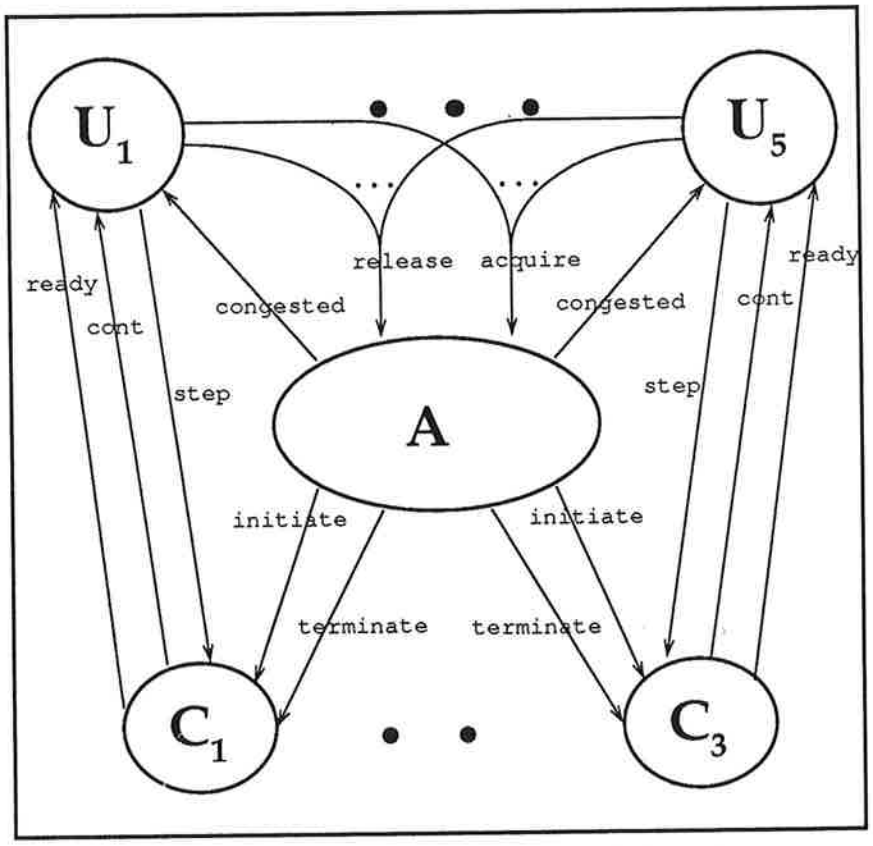

Figure 4: A schema of processes and channels

obtaining the access to the *Counter*, the *User* process can communicate directly to this process to obtain its services. Figure 4 shows the processes and channels they use to communicate.

When the *Allocator* receives a request from a *User*, it looks for a free *Counter*. If this is found, the *initiate* message is sent to it. This message contains the name of the *User*, so the *Counter* can send it the *ready* message. Then it waits to service the *User*. When all the *Counters* are busy, the *Allocator* sends the *congested* message to the *User*. Thus the *User* will retry the request until a *Counter* will be free.

Figures 5a, 5b and 5c show the code of the example processes. In this case the sequential language is C.

```
main()
(
 self      U1 : user;
 partners  A : allocator;
           C1, C2, C3 : counter;
 procvar   C (C1, C2, C3), X;
 channel from A type congested();
           from counter type ready();
           from C type cont = int;
           toward  A type acquire();
           toward  A type release(us = processname);
           toward all  counter type step();
 int end, count, i;

 for (;;){
   end = 1;
   send(A,acquire());
   rep
     boolean = end;
     receive(A,congested())
     docl
         end = 0;
     endcl;
   %
     boolean = end;
     receive(X:any of (C1,C2,C3), ready())
     docl
         for (i=0; i<10; i++)
             send(X,step());
         connect(C,X);
         send(A,release(X));
```

```
            receive(C,count);
            end = 0;
            detach(C);
            terminate;
        endcl;
    endrep;
  }
}
```

Figure 5a: The User process

```
main()
{
  self      A : allocator;
  partners  U1, U2, U3, U4, U5 : user;
            C1, C2, C3 : counter;
  procvar   COUNT[3], C, P;
  channel from user type acquire();
          from user type release(us = processname);
          toward all  counter type initiate(up = processname);
          toward all  counter type terminate();
  int end, counter[3], i;

  for (i=0;i<3;i++)
      counter[i] = 0;
  connect(COUNT[0],C1);
  connect(COUNT[1],C2);
  connect(COUNT[2],C3);
  rep
      receive (P:any of(user), acquire()) ;
      docl
          end = 1;
          i = 0;
          while (end && i<3) {
                if (counter[i] == 0) {
                    counter[i] = 1;
                    end = 0;
                    send (COUNT[i], initiate(P));
                }
                else   i++;
          }
          if (end)
                send (P, congested());
      endcl;
```

```
 %
    receive (P:any of(user), release(C));
    docl
        end  = 1;
        send (C, terminate()) ;
        i = 0;
         while (end && i<3) {
           if (COUNT[i] == C) (
                  counter[i] = 1;
                  end = 0;
            }
            else  i++;
         }
    endcl;
 endrep;
}
```

Figure 5b: The Allocator process

```
main()
{
 self      C1 : counter;
 partners  A : allocator;
           U1, U2, U3, U4, U5 : user;
 procvar   X, U (U1, U2, U3, U4, U5);
 channel from A type initiate(up = processname);
           from A type terminate();
           from U type step();
           toward all  user type cont =int;
           toward all  user type ready();
 int end, count;

 for (;;){
   end = 1;
   receive(A, initiate(X));
   connect(U,X);
   send(U,ready());
   rep
     boolean = end ;
     receive (U, step()) ;
     docl
        count++;
     endcl;
   % boolean  = end ;
     receive (A, terminate());
```

```
        docl
            send (U, count) ;
            end   = 0;
            count = 0;
              detach(U);
        endcl
      endrep;
    }
  }
```

Figure 5c: The Counter process

6. *Run-time Support*

The NERECO distributed run-time support has been implemented as a virtual machine by a set of cooperating processes located on the network nodes. They are UNIX processes communicating by means of lower level inter-process communication facilities. The distributed run-time support mainly implements the interpretation of the concurrent constructs of the language. Besides this, it provides for the initialization of the program and the logging of the concurrent constructs that are executed.

For process creation and communication, the UNIX system calls have been used, such as *fork*, *execl*, and *sockets*. In particular, *sockets* implement bidirectional communication channels among detached processes, without a common ancestor and even located on remote hosts.

Sockets provide the necessary support for local and remote communications in the NERECO run-time support. Furthermore, they also check for error conditions in the communications. In fact the use of *sockets* allows a process' partner which may be local or remote to be notified if the process has failed. Therefore *sockets* can be utilized as basic tools for the distributed handling of failures.

6.1 *Logic network*

When the user requests the execution of an application, some actions start from the node on which the execution request is performed, to set up an early set of channels towards the other nodes.

This phase is terminated when a logic network is completed, by building a complete connection among all the hosts.

The processes which support this phase are:

Nereco initializer (NRC)
: The NRC is the user interface component. The user invokes the NCR execution from the *shell* environment, specifying the program name. NRC delivers to the local RCSP process the execution request.

Remote Connection Service Point (RCSP)
: There is a RCSP process on each node, its address is constituted by a pair: (*host address, Internet port*). The RCSP receives the request from the NRC process and communicates with other RCSP processes on the remote nodes on that the processes must run. Then it *forks* and executes the LIS Master process, while each remote RCSP *forks* a LIS Slave process. After that, the RCSP breaks away from the current request and waits for other program execution requests.

Local Initializer Server (LIS)
: There are two kinds of LIS processes, Master and Slave. The LIS Master, which is located on the node where the execution is requested, tests the network set-up by communicating with the LIS Slaves that are located on the other nodes. LIS Master and LIS Slaves establish a logic mesh network, implemented by means of *sockets* (Figure 6). Each LIS Slave receives the name of the processes which will be executed on its node.

The NRC and the LIS processes are created dynamically for each distributed program, whereas the RCSP process is created in one single copy on each node when the bootstrap occurs, and it remains always active; in summary, it works like a UNIX *daemon*. Obviously, if the program is configured on only one node, no network procedure is executed and the program set-up is done only in that node.

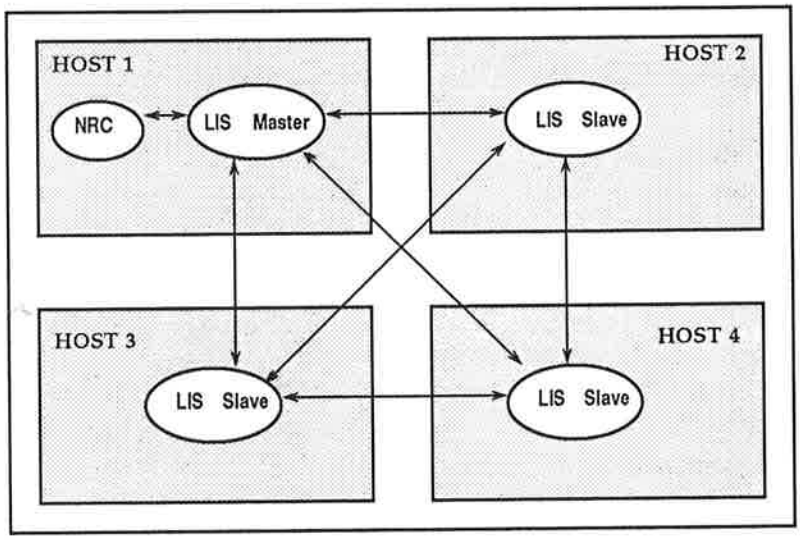

Figure 6: Connections among LIS processes

6.2 *Execution*

The Network Server is the logical component which provides for the creation and the run-time support of the concurrent constructs of the user processes. It implements the various forms of communication of the cooperation model, provides for the interpretation of the nondeterministic commands, and for the management of the dynamic channels.

The Network Server is implemented by the processes: NS, IN, OUT, and NETLOG. We denote the user processes which compose the program as Process Components (PCs) (Figure 7).

When the logic network set-up is terminated, the LIS Master process notifies the NRC process. Hence each LIS process is transformed into a NS process, by the *execl* system call.

Before the transformation, the LIS process creates on each node the IN and OUT processes. Every OUT process is connected to each IN process and vice versa. The existence of these processes makes it possible to enhance the computing bandwidth of the Network Server by executing in parallel external communications and internal computing.

On each node, the NS process loads its data structures with information about local PCs and remote PCs which are named in the communication constructs of local PCs. This information

allows the NS process to control the status of the local PCs and their remote partners.

The NS processes create local PCs by means of *fork* and *execl* system calls, hence each application process (PC) is mapped onto a UNIX process as child of the local NS process. After that, the NS processes set out to serve the requests coming from local PCs or from remote nodes. In the first case, the NS process will forward the communication requests to the remote hosts by the OUT process, or it will support the communications among the local PCs. Further, the NS processes cooperate with each other to maintain consistent information on the PCs' status.

To monitor the distributed application, the NETLOG process maintains a log of the executed concurrent construct. It is created only on the node where the execution has been requested, but it receives information from each node. Finally, the NETLOG stores in a file the information about the execution results of the concurrent constructs of the program.

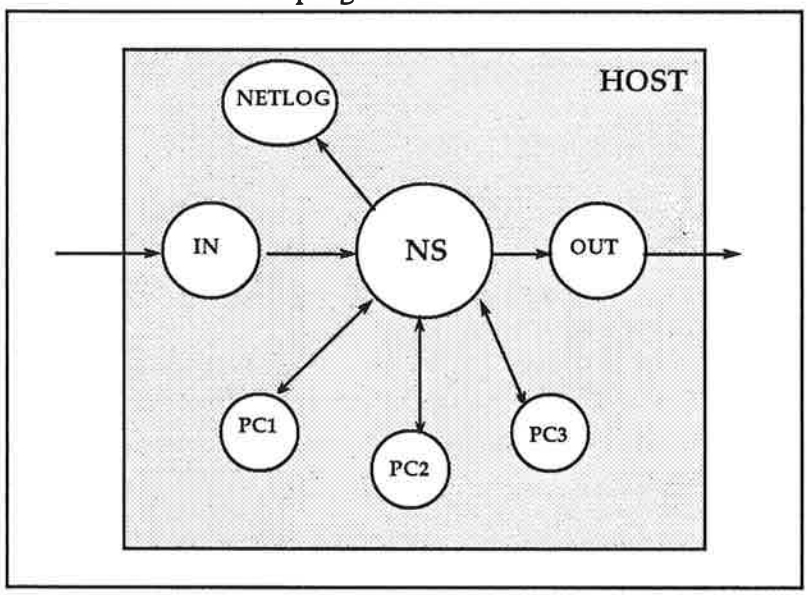

Figure 7: The PCs and the run-time support processes on one host

6.3 *Distributed termination*

The processes which compose the Network Server terminate when each PCs is terminated. Hence they provide to the Network Server termination.

On each host the NS process knows the state of local processes, that is which processes are running and which are terminated. If all local PCs are terminated, the NS process forces the termination of the local IN and OUT processes, then performs its termination. When all the run-time support processes on a node are terminated, it is notified to other nodes by the disconnection of sockets between the IN process and the remote OUT processes. The last node to terminate is the node in which the application was initiated and the NETLOG process was executing.

7. *Conclusion*

This paper has described the concurrent language and the tools of the NERECO system. This system is the distributed implementation of a message-passing concurrent model derived from the CSP model.

Using the NERECO system, a user can develop distributed applications constituted of a set of concurrent processes located on different nodes of the network. Each process carries out one of the program functionalities.

Using the tools offered by the language, the programmer can achieve many benefits in the development and testing of concurrent programs in comparison with the deficiencies of the traditional approaches.

The distributed concurrent language offers mechanisms to obtain location transparency, fault handling, modularity, and reliability.

Location transparency is provided because the concurrent language offers uniform communication mechanisms both between local and remote processes.

Fault handling can be implemented using the mechanisms offered by the language to handle explicitly the faults that can occur.

Modularity is implemented by isolating particular functionalities inside the single processes.

Reliability is provided by the language's strong checks at compile-time and static checks of consistency among the processes.

At present, NERECO is used to develop distributed programs. The system has proved that a high-level language is very useful in developing reliable distributed programs [Spezzano & Talia 1988].

References

Ada Joint Program Office, *Reference Manual for the Ada programming language*, ANSI/MIL-STD 1815 A, (1983).

A. V. Aho, R. Sethi and J. D. Ullman, *Compilers: Principles, Techniques and Tools*, Reading, MA: Addison-Welsey, 1986.

F. Baiardi, L. Ricci and M. Vanneschi, Stating checking of interprocess communication in ECSP, *ACM Sigplan Notices*, 19:290-299, 1984.

F. Baiardi, L. Ricci, A. Tomasi and M. Vanneschi, Structuring processes for a cooperative approach to fault-tolerant distributed software, *Proceedings 4th IEEE Symposium on Reliability in Distributed Software and Database Systems*, 1984a.

H. E. Bal, J. G. Steiner, A. S. Tanenbaum, Programming Languages for Distributed Computing Systems, *ACM Computing Surveys*, 21(3):261-322, 1989.

P. Brinch Hansen, *Operating System Principles*, Englewood Cliffs, NJ: Prentice-Hall, 1973.

P. Brinch Hansen, Joyce – A Programming Language for Distributed Systems, *Software – Practice and Experience*, 17:29-50, 1987.

D. Crookes and J. W. G. Elder, An experiment in language design for distributed systems, *Software – Practice and Experience*, 14:957-971, 1984.

L. DeFerrari, G. Spezzano, and D. Talia, NERECO: Architecture, *Technical Report*, CRAI, Rende, 1985.

N. H. Gehani, Broadcasting Sequential Processes (BSP), *IEEE Trans. on Software Engineering*, 10(4):343-351, 1984.

C. A. R. Hoare, Communicating Sequential Processes, *Communications of the ACM*, 21:666-677, 1978.

Inmos, *Occam Programming Manual*, Englewood Cliffs, NJ: Prentice-Hall, 1984.

M. Jazayeri et al., CSP/80: A language for communicating sequential processes, *IEEE Compcon Fall 1980 Conference Proceedings*, pages 736-740, 1980.

B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Englewood Cliffs, NJ: Prentice-Hall, 1978.

S. J. Leffler, R. S. Fabry and W. N. Joy, A 4.2BSD interprocess communication primer, *UNIX Programmer's Manual Berkeley Software Distribution*, Virtual VAX-11 Version, University of California, Berkeley, 1983.

B. Lyon et al., Overview Of The Sun Network File System, *Sun's Network File System Documentation*, Sun Microsystems, 1984.

J. Magee, J. Kramer and Sloman, The Conic support environment for distributed systems, *Proceedings of NATO Advanced Study Institute, Distributed Operating Systems: Theory and Practice*, Izmir, Turkey, 1986.

D. M. Ritchie and K. Thompson, The UNIX time-sharing system, *Bell System Technical Journal*, 57(6):1905-1929, 1978.

G. Spezzano, D. Talia and M. Vanneschi, NERECO: An Environment for the Development of Distributed Software, *EUUG Conference Proceedings*, pages 153-167, Dublin, Sept. 1987.

G. Spezzano and D. Talia, A Language Based Approach for Reliable Distributed Computing, *Proc. of IEEE Workshop on the Future Trends of Distributed Comp. Syst.*, pages 262-269, Hong Kong, Sept. 1988.

G. Spezzano and D. Talia, The Design of Fault-tolerant Distributed Software Using a Concurrent Language, *Proc. of 12th FTSD Int. Conference*, pages 260-265, Praga, Sept. 1989.

R. E. Strom and S. Yemini, NIL: An integrated language and system for distributed programming, *SIGPLAN Symposium on Programming Language Issues in Software Systems*, pages 73-82, 1983.

[*submitted Sept. 29, 1989; revised March 21, 1990; accepted May 16, 1990*]