```
}

/* function to print an error message and exit */
void error(str)
char *str;
{
  printf("Error: %s\n", str);
  exit(0);
}
```

Appendix B – Compilation stages of ELROS programs

Various stages in the compilation of an ELROS program is shown in figure 3. The ELROS program is first passed through the ELROS preprocessor which converts the ELROS statements to equivalent C code using sockets. The output of the ELROS preprocessor is then passed through a C compiler to get the final executable.

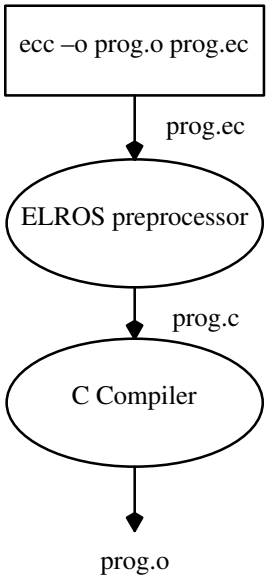

Figure 3 – Compilation stages of ELROS programs

Appendix C – Obtaining ELROS

The complete distribution of ELROS can be obtained by anonymous ftp from fram-sparc.ocf.llnl.gov in the directory "pub/elros". The distribution includes installation instructions, tutorials, and other documents on ELROS.

Appendix A – Contents of header files *include_elros.h and include_socket.h*

In this section we present the contents of ‘‘include” files used in our programs shown earlier.

A.1 Contents of *include_elros.h* file

include_elros.h is the header file used in our ELROS programs.

```
/* from include_elros.h  */
#include <stdio.h>

#define IP_ADDRESS ”130.18.208.91+” /* ip address of the server */
#define IP_PRT_LEN 25               /* ip_port address length */
#define NOD_LEN 3                   /* procs number length */
#define PRT_LEN 5                   /* port address length */
#define IP_LEN 20                   /* ip address length */
#define ERR_LEN 512                  /* length of the error message */
#define TIMEOUT 10                  /* time for which server blocks waiting for
                                       connection request from a client  */
#define TRUE 1
#define FALSE 0
#define NEW(obj, n) (obj *)malloc(n * sizeof(obj))
void error(str)                 /* print error message and exit */
char *str;
{
  printf(”%s\n”, str);
  exit(0);
}
```

A.2 Contents of *include_socket.h* file

include_socket.h is the header file used in our socket programs.

```
/*from  include_socket.h */
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define BUF_SIZ 25
#define NODES 4
#define NEW(obj, n) (obj *)malloc(n*sizeof(obj))

/* function to initialize the socket address structure */
void init_addr(address, port)
struct sockaddr_in *address;
int port;
{
   bzero((char *) address, sizeof(*address));
   address–>sin_family = AF_INET; /* Type of address */
   address–>sin_addr.s_addr = htonl(INADDR_ANY); /* IP address */
   /* INADDR_ANY specifies a
      wildcat address that
      matches any of the host’s
      IP address */
   address–>sin_port = htons(port); /* Protocol port number */
```

[3] Branstetter, M. L., J. A. Guse, D. M. Nessett, and L. C. Stanberry. 1992. Using ELROS To Implement ISO Applications, Lawrence Livermore National Laboratory, 1992.

[4] Branstetter, M. L., J. A. Guse, and D. M. Nessett. 1992. ELROS – An Embedded Language for Remote Operations Service, Lawrence Livermore National Laboratory, 1992.

[5] Comer, Douglas E. and David L. Stevens. 1993. Internetworking with TCP/IP, Volume III: Prentice Hall, 1993.

[6] Message Passing Interface Forum. Document for a Standard Message–Passing Interface. 1993. Technical Report, Technical Report No. CS–93–214., University of Tennessee, November 1993. Available on **netlib.**

[7] Skjellum, Anthony, Steven G. Smith, Nathan E. Doss, Alvin P. Leung, and Manfred Morari, The Design and Evolution of Zipcode, Parallel Computing, 1993. (Invited Paper, to appear).

[8] Skjellum, Anthony and Alvin P. Leung, Zipcode: A Portable Multicomputer Communication Library atop the Reactive Kernel, In Proceedings of the Fifth Distributed Memory Computing Conference (DMCC5), page 767–776. IEEE Press, April 1990.

[9] Stevens, W. Richard, UNIX Network Programming, Prentice Hall, 1990.

2. Each process calls *connect()* to connect to all the processes having process number smaller than this process, and thus establishes connection with those processes. Thus all processes get connected to all other processes in the group.
3. Each process calls *send()* to send a message (containing the process number of this process) to all other processes in the group.
4. Each process calls *recv()* to receive messages from all other processes in the group.
5. After receiving a message from a process, each process calls *close()* to disconnect all the sockets connected to other process and then terminate.

6 Future Work

- ELROS could be modified in such a way that its dependence on Berkeley sockets could be hidden so that different code generators and different environments could be used with ELROS.
- A collective interface using ELROS would be useful as we have discussed in this paper. To make ELROS meaningful for collective operations, we have to make a model of collective client–server computation (MPI intercommunicator do this [6]).
- An MPI code generator would be useful for ELROS version 2.1 and the one we have discussed in this paper.
- FORTRAN and C++ interfaces to ELROS version 2.1 and the one presented in this paper would be useful, but it involves a lot of work.
- Comparison of ELROS with other protocol compilers like RPCGEN and RPCGEN.NEW would be useful [1]. In particular, ELROS data type management appears much more powerful than RPCGEN. Also data conversion capabilities of ELROS depends on ASN.1 rather than the older XDR.

7 Conclusion

A few of the collective operations providing example functionality were presented, that ELROS does not provide. Socket versions for these operations were also presented for comparison. We observed that implementation of these operations using ELROS is much simpler than those using sockets. This is because of the wide range of data types (which we did not exploit) and good exception handling facilities supported by ELROS.

9. References

[1] Bloomer, John. 1991. Power Programming with RPC: O'Reilly & Associates, Inc..

[2] Branstetter, M. L., J. A. Guse, D. M. Nessett, and L. C. Stanberry. 1992. ELROS Primer, Lawrence Livermore National Laboratory, 1992.

```
    /* Total number of processes acting as servers to this process */
    servers = node_num;
    /* Total number of processes acting as clients to this process */
    clients = total_nodes – node_num – 1;
    /* Accept connection requests from all the clients */
    for(i = 0; i < clients; i++)
    {
        nodes[i+1+node_num] = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
        if(nodes[i+1+node_num] < 0)
            error("accept unsuccessful\n");
    }
    /* Compute the port number of the previous process to which this process should act as a client */
    previous_port_num = port_num – 1;
    /* For each of the processes that are already spawned, this process acts as a client. ie. Initiator */
    for(i = 0; i < servers; i++)
    {
        init_addr(&address, previous_port_num);
        /* Open TCP socket */
        if((nodes[node_num–i–1] = socket(AF_INET, SOCK_STREAM, 0)) < 0)
            error("server1: Can't open stream socket\n");
        /* Connect to server */
        if(connect(nodes[node_num–i–1], (struct sockaddr *) &address, sizeof(address)) < 0)
            error("****client: can't connect to server***\n");
        /* Decrement the value of the previous port number */
        previous_port_num = previous_port_num – 1;
    }
    /* copy the message to be sent from this port to string variable msg_out */
    sprintf(msg_out, "#########message from process%d##########", node_num);
    /* Pass message to each of the processes to which this process is bound */
    for(i = 0; i < total_nodes; i++)
        if(i != node_num)
        {
            send(nodes[i], msg_out, sizeof(msg_out), 0);
        }
    /* Get messages from each of the processes to which this process is bound */
    for(i = 0; i < total_nodes; i++)
        if(i != node_num)
        {
            recv(nodes[i], msg_in, sizeof(msg_in), 0);
            printf(" process%d: %s", node_num, msg_in);
            /* close the sockets */
            close(nodes[i]);
        }
    close(sockfd);
}
```

5.3.2.2 Description

The behavior of the above program can be summarized as follows:

1. Each process (except the process having the largest process number) will spawn its next higher numbered process, calls *Accept()* and blocks waiting for all the process having process numbers higher than this process to call *connect()* , after which this process establishes connection with all processes having process numbers higher than this process.

5.3.2 Socket Version

The socket version for the all_to_all scatter / gather operation is presented next.

5.3.2.1 Program Code

```
/* Implementation of all–to–all scatter / gather operation using sockets
 * USAGE:  a.out start_port_no. total_no._of_processes process_no.
 * e.g;          a.out 3050 5 1  –will start 5 processes
 */

#include ”include_socket.h”

main(argc, argv)
int argc;
char *argv[];
{
     int sockfd,              /* socket descriptor */
     clilen                   /* address length of client process */
     start_port_num,          /* port number of the root process */
     total_nodes,             /* total number of processes in the heap */
     node_num,                /* node number of this process */
     i,                       /* index variable */
     clients,                 /* number of processes acting as clients to this process */
     servers,                 /* number of processes acting as servers to this process */
     previous_port_num,       /* port number of the process that spawned this process */
     *nodes,                  /* array of sockets connected to all other processes */
     port_num;                /* port number of this process */
     char buff[BUF_SIZ],      /* buffer used for send and receive operations */
     next_node[NODES];        /* node_number of the next process to be spawned */
     struct sockaddr_in cli_addr, address;    /* Struct to hold an address */
     char msg_in[BUF_SIZ], msg_out[BUF_SIZ];
     start_port_num = atoi(argv[1]);
     total_nodes = atoi(argv[2]);
     node_num = atoi(argv[3]);
     port_num = start_port_num+node_num;   /* port number of this process */
     /* allocate memory to store the socket descriptors */
     if((nodes = NEW(int, total_processes)) == NULL)
          error(”Error allocating memory”);
     /* spawn another process if this is not the last process */
     if(node_num < total_nodes – 1)
     {
          sprintf(next_node, ”%i”, node_num+1);
          if(fork() == 0)
               if((execl(”./all_to_all”, ”all_to_all”, argv[1], argv[2], next_node, NULL)) == –1)
                    error(”Error using execl...\n”);
     }
     /* Open a TCP socket (an Internet stream socket). */
     if((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
          error(”server: Can’t open stream socket\n”);
     /* Initiate the address structure and fill in the server’s address */
     init_addr(&address,port_num);
     /* Bind the address structure to the socket */
     if(bind(sockfd, (struct sockaddr *) &address, sizeof(address)) < 0)
          error(”server: Can‘t bind local address\n”);
     listen(sockfd, 5);
```

```
        if(i != process_num)
        {
            procs = process[i];
            Accept procs
               {
                 On pass_message(msg_in)
                    {
                      printf(" process%d: %s", process_num, msg_in);
                    }
            }
            /* Enter the Exceptions clause in case the accept command fails */
            Exceptions
               {
                 On Done
                    {
                      printf("process%d: Unbinding process%d", process_num, i);
                      /* unbind the process for which the accept command failed */
                      Unbind(procs);
                    }
            }

            /* unbind the process after receiving the message */

            Unbind(procs);

        }
}
```

5.3.1.2 Description

The functionality of the above program can be summarized as follows:

1. Each process (except the process having the largest process number) will spawn its next higher numbered process, calls *Bind_addr()* as a responder (server) and blocks waiting for all the process having process numbers higher than this process to call *Bind_addr()* as clients to this process, after which this process establishes connection with all processes having process numbers higher than this process.
2. Each process calls *Bind_addr()* as an initiator (client) to all the processes having process number smaller than this process, and thus establishes connection with those processes. Thus all processes get connected to all other processes in the group.
3. Each process executes *Give* statements on the *pass_message* operation to send a message (containing the process number of this process) to all other processes in the group.
4. Each process executes *Accept* statements on *pass_message* operation to receive messages from all other processes in the group.
5. After receiving a message from a process, each process calls *Unbind()* to disconnect from that process and then terminate.

```
process */
responders = total_processes – process_num – 1;
/* Port number of this process is the sum of the starting port number and the port number of this port */
port_num = process_num + start_port_num;
/* If the number of active processes are less than total num of processes, spawn another
process */
if(process_num < total_processes – 1)
{
     sprintf(next_proc, ”%i”, process_num+1);
     if(fork() == 0)
          if((execl(”./all_to_all”, ”all_to_all”, argv[1], argv[2], next_proc, NULL)) == –1)
               error(”Error using execl...”);
}
/* For each of the processes to be created, this process acts as a server ie. Responder */
for(i = 0; i < responders; i++)
{
     sprintf(ip_port_num,”%s%d”, IP_ADDRESS, port_num);
     /* Initiate the address structure and fill in the server’s address */
     init_puresocket_addr(&addr,  Responder, ip_port_num);
     /* Establish communication with the client */
     Bind_addr(procs, Asn_1_ber, &addr, 20);
     /* Copy the instance variable which is bound to an array */
     process[i+1+process_num] = procs;
}
/* Compute the port number of the previous process to which this process should act as a
client ie. Initiator */
previous_port_num = port_num – 1;
/* For each of the processes that are already created, this process acts as a client. ie.
     Initiator */
for(i = 0; i < initiators; i++)
{
     /* Determine the IP–PORT address of the previous process */
     sprintf(previous_process_addr,”%s%d”, IP_ADDRESS,previous_port_num);
     /* Initialize the address structure and fill in the server’s address */
     init_puresocket_addr(&addr,  Initiator, previous_process_addr);
     /* Establish communications with the server */
     Bind_addr(procs, Asn_1_ber, &addr, 20);
     /* Copy the instance variable which is bound to an array */
     process[process_num–i–1] = procs;
     /* Decrement the value of the previous port number */
     previous_port_num = previous_port_num – 1;
}
/* copy the message to be sent from this port to string variable msg_out */
sprintf(msg_out, ”##########message from process%d##########”, process_num);
/* Pass message to each of the processes to which this process is bound */
for(i = 0; i < total_processes; i++)
     if(i != process_num)
     {
          procs = process[i];
          Give procs
             {
               pass_message(msg_out);
          }
     }
/* Get messages from each of the processes to which this process is bound */
for(i = 0; i < total_processes; i++)
```

```
Interface all_to_all;
    PrintableString(STR_LEN) message;

    Invoke  pass_message OP_NO
        message;
End;
```

5.3.1.1 Program Code

```
/* Implementation of all–to–all scatter / gather operation using ELROS
 * USAGE:  a.out start_port_no. total_processes process_no
 * e.g;    a.out 1050 5 1  –will start 5 processes, assigning port 1050 to process #0,
 *         port 1051 assigned to process  #1 and so on and messages are exchanged between all the
 *         processes.
 */

#include ”include_elros.h”
#include ”interfaces/all_to_all.i”   /* Import the all_to_all interface */

main(argc, argv)
int argc;
char *argv[];
{
    int   total_processes,      /* total no. of processes */
    start_port_num,             /* port no. of process #1 */
    parent_port_num,            /* port no. of the parent process */
    process_num,                /* process no. of this process */
    port_num,                   /* port no. of this process */
    previous_port_num,          /* port no.of the previous process */
    initiators,                 /* no. of processes for which this process is a initiator */
    responders,                 /* no. of processes for which this process is a  responder */
    i;                          /* index variable */
    char    ip_port_num[IP_PRT_LEN],      /* IP+port number of this process */
    previous_process_addr[IP_PRT_LEN],  /* IP+port number of the parent process */
    next_proc[NOD_LEN],                 /* address of the next process */
    string_variable[ERR_LEN];           /* string to store error message */
    /* Declare the instance variables */
    Control(all_to_all) procs, *process;
    /* Declare the address structure variable */
    Bind_address addr;
    /* Declare the pass_message operation argument */
    message msg_in, msg_out;
    /* Allocate memory for instance variables */
    if((process = NEW(Control(all_to_all, total_processes))) == NULL)
        error(”Error allocating memory\n”);
    /* Assign the starting port number to the integer variable start_port_num */
    start_port_num = atoi(argv[1]);
    /* Assign the total number of process to the integer variable total_processes */
    total_processes = atoi(argv[2]);
    /* Assign the process number to the integer variable process_num */
    process_num = atoi(argv[3]);
    /* Total number of processes for which this process acts as client ie. Initiates is equal to the
    process number of this process */
    initiators = process_num;
    /* Total number of processes for which this process acts as server ie. Responds is equal to
    the difference between the total number of processes and the process number of this
```

```
        }
      }
/* Close all open sockets */
close(childfd1);
close(childfd2);
close(sockfd);
close(sockfd1);
}
/* function to calculate largest of three numbers */
int biggest(n1, n2, n3)
int n1, n2, n3;
{
  big = (n1 > n2) ? n1 : n2;
  big = (n3 > big) ? n3 : big;
  return(big);
}
```

5.2.2.2 Description

The collapse program presented above works as follows:

1. Processes are spawned and connected in exactly the same manner as was done in our broadcast program.
2. Processes at the leaves send their process number to its parent process by calling the *send()*. (This is an example calculation).
3. All other processes except the processes at the leaves and the root process (process #1) receives the process number of their child processes by calling *recv()*, computes the maximum of these numbers and its own process number, and passes this maximum number to its parent process.
4. The root process receives two numbers from its child processes and computes the maximum of all the processes numbers in the tree.
5. All the processes disconnect from their child processes by calling *close()* and then terminate.

5.3 All–to–All Scatter / Gather

The program we present next produces the desired number of processes, each directly connected to all other processes. This structure can be used when each process in the group has a message to be sent to all other processes in the group.

5.3.1 ELROS Version

The interface definition for the ELROS program is shown below. The interface definition has one operation named *pass_message* that has a character string as argument. This operation is called by all the processes to broadcast a character string to all other processes in the group.

```
#define STR_LEN 50
#define OP_NO 10
```

```
      /*
       * Wait for connection from a client process.
       * This is an example of a concurrent server.
       */
      if(node_num*2 <= total_nodes)
        {
          childfd1 = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
          if(childfd1 < 0)
            error("accept unsuccessful\n");
          }
      if(node_num*2 < total_nodes)
        {
          childfd2 = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
          if(childfd2 < 0)
            error("accept unsuccessful\n");
          }
 if(node_num != 1)
   {
/* calculate the port number of  the parent process */
      parent_port = start_port_num + node_num/2 – 1;
      /* Initialize the addr structure of the server */
      init_addr(&address, parent_port);
      /* Open TCP socket */
      if((sockfd1 = socket(AF_INET, SOCK_STREAM, 0)) < 0)
          error("server1: Can't open stream socket\n");
      /* Connect to server */
      if(connect(sockfd1, (struct sockaddr *) &address, sizeof(address)) < 0)
         error("****client: can't connect to server***\n");
}
      /********************** COLLAPSE OPERATION ****************************/
      /* if this is the leaf node, send its node number to its parent process */
      if(node_num*2 > total_nodes)
        {
          sprintf(buff, "%d", node_num);
          send(sockfd1, buff, sizeof(buff), 0);
        }
      /*  if this is not the leaf node, receive numbers from its child processes */
      else
        {
          while(recv(childfd1, buff, sizeof(buff), 0) < 0)
            printf("client cannot receive. Trying again .....\n");
          number1 = atoi(buff);
          if(node_num*2 < total_nodes)
            while(recv(childfd2, buff, sizeof(buff), 0) < 0)
              printf("client cannot receive. Trying again .....\n");
          number2 = atoi(buff);
          sprintf(buff, "%d", biggest(number1, number2, node_num));
         /* if this is the root process, just print the maximum value */
          if(node_num == 1)
            printf("Root: maximum = %s\n", buff);
            / * if this is not the root process, send the maximum value to its parent process */
          else
            {
              printf("ndoe%d: maximum = %s\n", node_num, buff);
              send(sockfd1, buff, sizeof(buff), 0);
```

```
 * eg;          a.out 3050 5 1  –will start 5 processes
 */

#include ”include_socket.h”

main(argc, argv)
int argc;
char *argv[];
{
   int sockfd,          /* socket descriptor */
     sockfd1,           /* socket connected to the parent process */
     clilen,                 /* address length of child process */
     childfd1,               /* socket connected to the first child process */
     childfd2,               /* socket connected to the second child process */
     parent_port,            /* port number of the parent process */
     i,                      /* index variable */
     number1,                /* number received from the first child */
     number2,                /* number received from the second child */
     start_port_num,         /* port number of the root process */
     total_nodes,            /* total number of processes in the heap */
     node_num;               /* node number of this process */
     short port;             /* port number of this process */
   char buff[BUF_SIZ],   /* buffer used for send and receive operations */
     child_node[NODES];                  /* child process */
   struct sockaddr_in cli_addr, address;  /* Struct to hold an address */
   extern void init_addr();
   extern void error();
     int biggest();
     start_port_num = atoi(argv[1]);
     total_nodes = atoi(argv[2]);
     node_num = atoi(argv[3]);
     if(node_num*2 < total_nodes)          /* spawn two child processes */
       {
         for(i = 0; i < 2; i++)
           {
            sprintf(child_node, ”%d”, node_num*2+i);
            if(fork() == 0)
              execl(”./collapse”, ”collapse”, argv[1], argv[2], child_node, NULL);
           }
       }
     else if(node_num*2 == total_nodes)       /* spawn one child process */
       {
         sprintf(child_node, ”%d”, node_num*2);
         if(fork() == 0)
           execl(”./collapse”, ”collapse”, argv[1], argv[2], child_node, NULL);
       }
     /* Open a TCP socket (an Internet stream socket). */
     if((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
          error(”server: Can’t open stream socket\n”);
     /* Initialize the address structure for this node */
     port = start_port_num + node_num – 1;  /* port number of this node */
     init_addr(&address, port);                  /* Initialize the address structure */
     /* Bind our local address so that the client can send to us. */
     if(bind(sockfd, (struct sockaddr *) &address, sizeof(address)) < 0)
          error(”server: Can‘t bind local address\n”);
     listen(sockfd, 5);
```

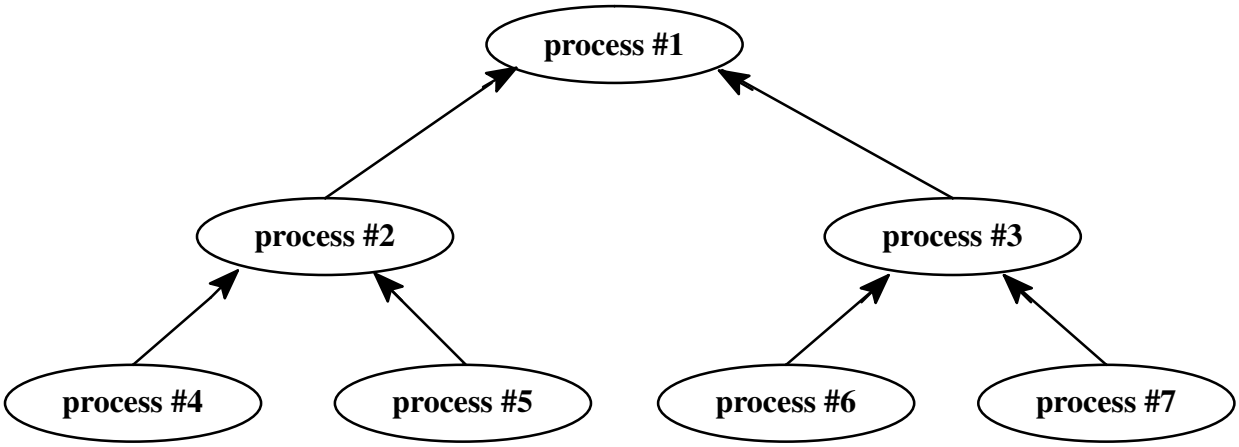

Figure 3 – Illustration of collapse operation for a group of seven processes interconnected in the form of a binary tree

The collapse operation in a group of seven processes interconnected in the form of a binary tree is shown in figure 3. Our collapse program using ELROS works as follows:

1. Processes are spawned and connected in exactly the same manner as was done in our broadcast program.
2. Processes at the leaves send their process numbers to their parent processes by executing the *give* statement on the *compute_maximum* operation.
3. All other processes except the processes at the leaves and the root process (process #1) receives the process number of their child processes by executing the *Accept* statement on the *compute_maximum* operation, computes the maximum of these numbers and its own process number, and passes this maximum number to its parent process.
4. The root process receives two numbers from its child processes and computes the maximum of all the processes numbers in the tree.
5. All the processes disconnect from their child processes by calling *Unbind()* and then terminate.

5.2.2 Socket Version

Next we present a program that uses sockets to implement the collapse operation. This program works the same way as the collapse program we presented earlier that used ELROS.

5.2.2.1 Program Code

```
/* Implementation of the collapse operation using sockets      */
/* USAGE:  a.out start_port_no. total_processes process_no.
```

```
        /* Unbind the first child process */
        Unbind(procs);
        if(proc_num*2 != total_processes)
        {
                procs = process[proc_num*2+1];
                /* Accept number form the second child if it exists */
                Accept procs
                {
                  On compute_maximum(number2)
                     {
                       printf(" process%d: number2 = %d", proc_num, number2);
                     }
              }
              /* Unbind the second child process */
              Unbind(procs);
              /* compute the maximum of the numbers received from the child processes */
              (*compute_max_fun_ptr)(&number1, &number2, &proc_num);
        }

        /* If there is only one child process, assign the number received by that process to max */

        /* If the process is not the root process, pass the computed maximum value to its parent process */
        if(proc_num != 1)
        {
              procs = process[proc_num/2];
              Give procs
                 {
                   compute_maximum(number1);
              }
        }
        else
              printf("process%d: maximum = %d\n", proc_num, number1);
    }
    /* If the process is the leaf process, pass its process number to its parent process */
    else
    {
        procs = process[proc_num/2];
        Give procs
           {
             compute_maximum(proc_num);
        }
    }
}

/* function to find the maximum of three numbers */
void compute_maximun_fun(number1, number2, number3)
NUMBER *number1, *number2, *number3;
{
    *number1 = (*number1 < *number2) ? *number2 : *number1;
    *number1 = (*number1 < *number3) ? *number3 : *number1;
}
```

5.2.1.2 Description

```
    if(fork() == 0)
        if((execl("./collapse", "collapse", argv[1], argv[2], child_proc, NULL)) == –1)
            error("Error using execl.....");
}
    /* Initiate the address structure and fill in the server's address */
    init_puresocket_addr(&addr,  Responder, ip_port_num);
    /* Establish communication with the client */
    Bind_addr(procs, Asn_1_ber, &addr, total_processes);
    process[proc_num*2] = procs;
}
/* Establish connection with parent process */
/* All processes except the root process will have parent process */
if(proc_num != 1)
{
    sprintf(parent_process_addr,"%s%d",IP_ADDRESS,parent_port_num);
    init_puresocket_addr(&addr,  Initiator, parent_process_addr);
    /*  Execute Bind_address until communication is established with the server */
    do
      {
       flag = FALSE;
      Bind_addr(procs, Asn_1_ber, &addr, total_processes)
      Exceptions
      {
        On Com_error(status, string_variable)
          {
            printf("process%d: Bind_addr error– %s", proc_num, string_variable);
            flag = TRUE;
        }
      }
    }
    while(flag == TRUE);
}
/* copy the instance variable procs to the array process */
process[proc_num/2] = procs;
/* If the process is not a leaf process, accept the numbers from the child processes */
if(proc_num*2 <= total_processes)
{
    procs = process[proc_num*2];
     flag = FALSE;
    /* Accept number form the first child */
    Accept procs
      {
        On compute_maximum(number1)
          {
                printf(" process%d: number1 = %d", proc_num, number1);
        }
    }
    Exceptions
    {
        On Done
        {
            printf(" process%d: Unbinding process%d", proc_num, proc_num*2);
            flag = TRUE;
        }
    }
```

```
int   total_processes,      /* total no. of processes */
start_port_num,             /* port no. of process #1 */
parent_port_num,            /* port no. of the parent process */
port_num,                   /* port no. of this process */
i,                          /* index variable */
status;                     /* error status */
short     flag;             /* reset on a communication error */
short bound;
char      ip_port_num[IP_PRT_LEN],      /* IP+port number of this process */
parent_process_addr[IP_PRT_LEN],        /* IP+port number of the parent process */
child_proc[NOD_LEN],                    /* node no. of child process */
string_variable[ERR_LEN];               /* string to store error message */

void compute_maximun_fun();
void (*compute_max_fun_ptr)();
/* Declare the instance variable */
Control(all_to_all) procs, *process;
/* Declare the address structure variable */
Bind_address addr;

/* Declare variables of type NUMBER, which is of the ELROS data type Integer */
NUMBER number1, number2, proc_num;
compute_max_fun_ptr = compute_maximun_fun;
/* Allocate memory for instance variable */
if((process = NEW(Control(all_to_all),  total_processes)) == NULL)
     error(" Error allocating memory for Instance variable\n");
/* Assign the starting port number to the integer variable start_port_num */
start_port_num = atoi(argv[1]);
/* Assign the total number of processes to the integer variable total_processes */
total_processes = atoi(argv[2]);
/* Assign the process number to the integer variable proc_num */
proc_num = atoi(argv[3]);
/* Compute the IP_PORT number of this process */
sprintf(ip_port_num, "%s%d", IP_ADDRESS, start_port_num+proc_num);
/* Calculate the port number of the parent process */
parent_port_num = (start_port_num+proc_num/2);
/* if the number of children to the process are two, spawn the two processes */
if(proc_num*2 < total_processes)
{
     for(i = 0; i < 2; i++)
     {
          sprintf(child_proc, "%i", proc_num*2+i);
          if(fork() == 0)
               if((execl("./collapse", "collapse", argv[1], argv[2], child_proc, NULL)) == –1)
                    error("Error using execl.....");
          /* Initiate the address structure and fill in the server's address */
          init_puresocket_addr(&addr,  Responder, ip_port_num);
          /* Establish communication with the client */
          Bind_addr(procs, Asn_1_ber, &addr, total_processes);
          process[proc_num*2+i] = procs;
     }
}
/* if the number of children to the process is one, span the child process */
else if(proc_num*2 == total_processes)
{
     sprintf(child_proc, "%i", proc_num*2);
```

3. Processes at the leaves calls *connect()* and get connected to their parent process.
4. Parent processes of the processes at the leaf calls *connect()* and get connected to their parent process (if they have one) after establishing connection with their child processes as servers.
5. Thus all the processes get connected to their parent and child processes forming a binary tree structure of processes
6. The root process (process #1) sends a message (a character string) to its child processes which, in turn propagate the message to their child processes. This way the message send (broadcast) by process #1 is received by all the other processes in the tree structure.
7. All processes calls *close()* to close the sockets that were connected to other processes.

5.2 Collapse Operation

The Binary trees can also be used to implement collapse operations as shown in figure 3. Our ELROS Program for collapse operation is shown below.

5.2.1 ELROS Version

Our implementation of the collapse operation using ELROS uses the same structure of processes connected in the form of a binary tree that was used to implement the broadcast operation. Interface definition for the collapse program is shown below: The operation *compute_maximum* is used by the clients to send a number. The server determines the maximum of the numbers it received and sends this maximum number to its server.

```
#define NO_OCTS 4
#define OP_NO   10
Interface all_to_all;
    Integer(NO_OCTS) NUMBER;
    Invoke   compute_maximum OP_NO
        NUMBER;
End;
```

5.2.1.1 Program Code

```
/* Implementation of the collapse operation using ELROS                */
/* USAGE: ecc –o collapse collapse.ec
*          collapse start_port_no. total_no._of_processes process_no.
* eg;      collapse 3050 5 1  –will start 5 processes
*/

#include ”include_elros.h”
#include ”interfaces/collapse.i”

main(argc, argv)
int argc;
char *argv[];
{
```

```
    if(node_num != 1)
    {
        /* Initialize the addr structure of the server */
        parent_port = start_port_num + node_num/2 – 1; /* port number of */
        /* the parent process */
        init_addr(&address, parent_port);
        /* Open TCP socket */
        if((sockfd1 = socket(AF_INET, SOCK_STREAM, 0)) < 0)
            error("server1: Can't open stream socket\n");
        /* Connect to server */
        if(connect(sockfd1, (struct sockaddr *) &address, sizeof(address)) < 0)
            error("****client: can't connect to server***\n");
    }
    if(node_num == 1)
    {
        strcpy(buff, "** msg from node1 **");
        if(node_num*2  <= total_nodes)
            send(childfd1, buff, sizeof(buff), 0);
        if(node_num*2  <  total_nodes)
            send(childfd2, buff, sizeof(buff), 0);
    }
    else
    {
        while(recv(sockfd1, buff, sizeof(buff), 0) < 0)
            printf("client cannot receive\n");
        printf("node%d:  %s\n", node_num, buff);
        if(node_num*2  < = total_nodes)
            send(childfd1, buff, sizeof(buff), 0);
        if(node_num*2  <  total_nodes)
            send(childfd2, buff, sizeof(buff), 0);
    }
 /* Close all open sockets */
    close(sockfd);
    close(sockfd1);
    if(node_num*2 <= total_nodes)
        close(childfd1);
    if(node_num*2 < total_nodes)
        close(childfd2);
}
```

5.1.2.2 Description

The above program works as follows:

1. Process #1 (the root process) calculates the number of child processes it is supposed to have (either one or two) and spawn that many child processes. Process #1 calls *Accept()* twice and blocks until its child processes calls *connect()* to establish a connection with this process.
2. Similarly all the other processes (except the processes at the leaves) spawn their child processes and blocks after calling *Accept()* until their child processes executes *connect()* to connect to these processes.

```
  i,                        /* index veriable */
  start_port_num,       /* port number of the root process */
  total_nodes,            /* total number of processes in the heap */
  node_num;              /* node number of this process */
  short port;              /* port number of this process */
char buff[BUF_SIZ],   /* buffer used for send and receive operations */
  child_node[NODES];     /* child process */
struct sockaddr_in cli_addr, address;    /* Struct to hold an address */
extern void init_addr();
extern void error();
  start_port_num = atoi(argv[1]);
  total_nodes = atoi(argv[2]);
  node_num = atoi(argv[3]);
  if(node_num*2 < total_nodes)   /* spawn two child processes */
  {
       for(i = 0; i < 2; i++)
       {
            sprintf(child_node, "%d", node_num*2+i);
            /* child process no. as a */
            /*    character string   */
            if(fork() == 0)
                 execl("./broadcast", "broadcast", argv[1], argv[2], child_node, NULL);
       }
  }
  else if(node_num*2 == total_nodes)     /* spawn one child process */
  {
       sprintf(child_node, "%d", node_num*2);
       if(fork() == 0)
            execl("./broadcast", "broadcast", argv[1], argv[2], child_node, NULL);
  }
  /* Open a TCP socket (an Internet stream socket). */
  if((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
       error("server: Can't open stream socket\n");
  /* Initialize the address structure for this node */
  port = start_port_num + node_num – 1;  /* port number of this node */
  init_addr(&address, port);           /* Initialize the address structure */
  /* Bind our local address so that the client can send to us. */
  if(bind(sockfd, (struct sockaddr *) &address, sizeof(address)) < 0)
       error("server: Can't bind local address\n");
  listen(sockfd, 5);
  /*
   * Wait for a connection from a client process.
   * This is an example of a concurrent server.
   */
  if(node_num*2 <= total_nodes)
  {
       childfd1 = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
       if(childfd1 < 0)
            error("accept unsuccessful\n");
  }
  if(node_num*2 < total_nodes)
  {
       childfd2 = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
       if(childfd2 < 0)
            error("accept unsuccessful\n");
  }
```

1. Process #1 (the root process) calculates the number of child processes it is supposed to have (either one or two) and spawn that many child processes. Process #1 calls *Bind_address()* twice specifying its role as a server (Responder) and blocks until its child processes calls *Bind_addr()* as clients (Initiators) to this process.
2. Similarly all the other processes (except the processes at the leaves) spawn their child processes and block after calling *Bind_addr()* until their child processes execute *Bind_addr()* as clients to these processes.
3. Processes at the leaf calls *Bind_addr()* as a client to their parent process and thus get connected to their parent process.
4. Parent processes of the leaf processes calls *Bind_addr()* as clients to their parent process (if they have one) after establishing connection with their child processes as servers.
5. Thus all the processes get connected to their parent and child processes forming a binary tree structure of processes
6. The root process (process #1) sends a message (a character string) to its child processes, which in turn propagate the message their children. This way the message send (broadcast) by process #1 is received by all the other processes in the tree structure.
7. All processes calls *Unbind()* to disconnect from their children processes and the processes are terminated.

5.1.2 Sockets Version

The broadcast program presented earlier used ELROS. We rewrote the same program using sockets. The program code for the socket version is shown below.

5.1.2.1 Program Code

```
/* Implementation of the broadcast operation using sockets
 * USAGE: cc –o broadcast broadcast.c
 *        broadcast start_port_no. total_no._of_processes process_no.
 * e.g;   broadcast 3050 5 1  –will start 5 processes
 */

#include ”include_socket.h”

main(argc, argv)
int argc;
char *argv[];
{
   int sockfd,          /* socket descriptor */
     sockfd1,           /* socket connected to the parent process */
     clilen,            /* address length of child process */
     childfd1,          /* socket connected to the first child process */
     childfd2,          /* socket connected to the second child process */
     parent_port,       /* port number of the parent process */
```

```
    if(process_num*2  <= total_processs)
    {
        procs = process[process_num*2];
        Give procs
          {
            pass_message(msg);
        }
        Unbind(procs);
    }
    /* If the process is the root process is the root process, send its message to its second child processes if it
has one. If it is not, send the message accepted by its parent process to its second  child processes if it has
one*/
    if(process_num*2+1  <= total_processs)
    {
        procs = process[process_num*2+1];
        Give procs
          {
            pass_message(msg);
        }
        Unbind(procs);
    }
}
```

5.1.1.2 Description

The broadcast operation in a group of processes connected in the form of a binary tree is shown in figure 2. Our ELROS program for broadcast operation works as follows:

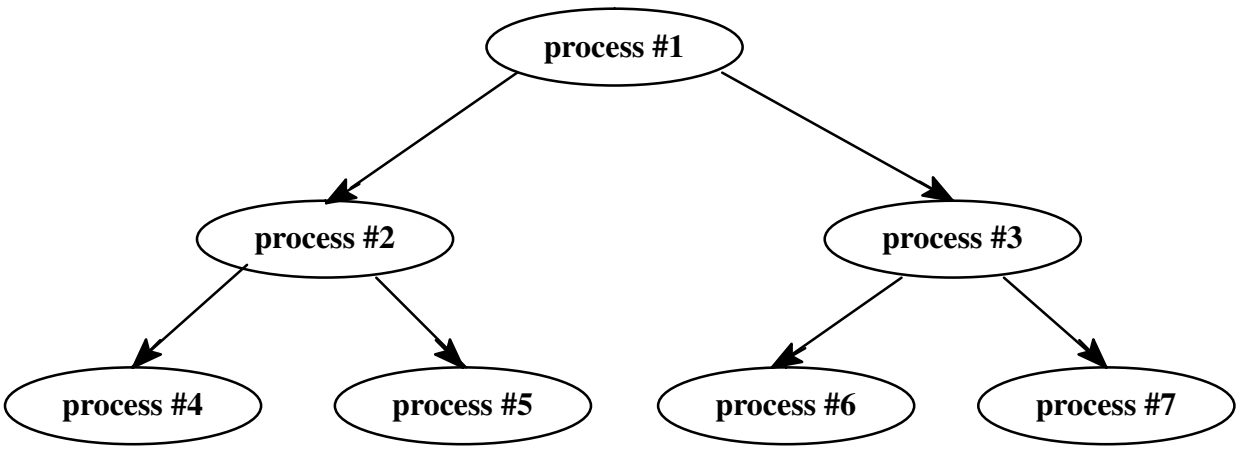

Figure 2 – Illustration of broadcast operation for a group of seven processes interconnected in the form of a binary tree

```
                error("Error using execl.....");
        /* Initiate the address structure and fill in the server's address */
        init_puresocket_addr(&addr,  Responder, ip_port_num);
        /* Establish communication with the client */
        Bind_addr(procs, Asn_1_ber, &addr, total_processs);
        process[process_num*2] = procs;
    }
    /* Establish connection with parent process */
    /* All processes except the root process will have parent process */
    if(process_num != 1)
    {
        sprintf(parent_process_addr,"%s%d",IP_ADDRESS,parent_port_num);
        init_puresocket_addr(&addr,  Initiator, parent_process_addr);
        /*  Execute Bind_address until communication is established with the server */
        do
        {
            flag = FALSE;
            Bind_addr(procs, Asn_1_ber, &addr, total_processs)
              Exceptions
              {
                On Com_error(status, string_variable)
                  {
                    printf("process%d: Bind_addr error– %s", process_num, string_variable);
                    exit(0);
                    flag = TRUE;
                  }
            }
        }
      while(flag == TRUE);
    /* copy the instance variable procs to the array process */
    process[process_num/2] = procs;
    }
    /* if this is the root process, copy the message to be sent to the string variable msg */
    if(process_num == 1)
    sprintf(msg, "##########message from process%d##########", process_num);
    /* If the process is not the root process, accept message from the parent process */
    if(process_num != 1)
    {
        procs = process[process_num/ 2];
        Accept procs
          {
            On pass_message(msg)
              {
                printf("process%d: %s", process_num, msg);
            }
        }
        Exceptions
          {
            On Done
              {
                printf("process%d: Could not accept from process%d", process_num, process_num/2);
            }
        }
    }
```

/* If the process is the root process, send its message to its first child processes. If it is not, send the message accepted by its parent process to its first child processes */

```
{
    short bound;
    int   total_processs,       /* total no. of processes */
    process_num,                /* process no. of this process */
    start_port_num,             /* port no. of process #1 */
    parent_port_num,            /* port no. of the parent process */
    port_num,                   /* port no. of this process */
    i,                          /* index variable */
    status;                     /* error status */
    short     flag;             /* reset on a communication error */
    char      ip_port_num[IP_PRT_LEN],      /* IP+port number of this process */
    parent_process_addr[IP_PRT_LEN],        /* IP+port number of the parent process */
    child_proc[NOD_LEN],                    /* node no. of child process */
    string_variable[ERR_LEN];               /* string to store error message */
    /* Declare the instance variables */
    Control(broadcast) procs, *process;
    /* Declare the address structure variable */
    Bind_address addr;
    /* Declare the pass_message operation argument */
    message msg;
    /* Allocate memory for instance variable */
    if((process = NEW(Control(broadcast), total_processs)) == NULL)
        error("Error allocating memory\n");
    }
    /* Assign the starting port number to the integer variable start_port_num */
    start_port_num = atoi(argv[1]);
    /* Assign the total number of processes to the integer variable total_processes */
    total_processs = atoi(argv[2]);
    /* Assign the process number to the integer variable process_num */
    process_num = atoi(argv[3]);
    /* Compute the IP_PORT number of this process */
    sprintf(ip_port_num, "%s%d", IP_ADDRESS, start_port_num+process_num);
    /* Calculate the port number of the parent process */
    parent_port_num = (start_port_num+process_num/2);
    /* if the number of children to the process are two, spawn the two processes */
    if(process_num*2 < total_processs)
    {
        for(i = 0; i < 2; i++)
        {
            sprintf(child_proc, "%i", process_num*2+1);
            if(fork() == 0)
                if((execl("./broadcast", "broadcast", argv[1], argv[2], child_proc, NULL)) == –1)
                    error("Error using execl");
            /* Initiate the address structure and fill in the server's address */
            init_puresocket_addr(&addr,  Responder, ip_port_num);
            /* Establish communication with the client */
            Bind_addr(procs, Asn_1_ber, &addr, total_processs);
            process[process_num*2+i] = procs;
        }
    }
    /* if the number of children to the process is one, span the child process */
    else if(process_num*2 == total_processs)
    {
        sprintf(child_proc, "%i", process_num*2);
        if(fork() == 0)
            if((execl("./b", "b", argv[1], argv[2], child_proc, NULL)) == –1)
```

collective operations such as broadcast, collapse, and all–to–all scatter / gather operations [6, 7]. The collapse operations is used to gather some data from all the processes in the group to a single process (the root process). The all–to–all scatter / gather operation is used to exchange messages between all the processes in the group. First we present the ELROS version and then the socket version of implementing these operations. To implement the broadcast and collapse operations we used processes connected in the form of binary tree structure rather than having a direct connection between all the processes in the group and the root process. Although this results in a slight communication overhead in terms of the number of hops the message has to take, the total communication time and the total number of connections was considerably reduced. We used a fully connected process structure to implement the all–to–all scatter / gather operation.

5.1 Broadcast Operation

The broadcast operation is used to broadcast a message from one member of the group (the root process) to all other members of the group. In our example, we pass a character string from the root process to all other processes in the group.

5.1.1 Broadcast With ELROS

Next we present the program code to implement the broadcast operation using ELROS. The interface definition for this program is shown below.

```
/* FILE: broadcast.i */
/* Interface definition for the broadcast program */
#define STR_LEN 50
#define OP_NO 10
Interface broadcast;
    PrintableString(STR_LEN) message;
    Invoke  pass_message OP_NO
        message;
End;
```

5.1.1.1 Program Code

```
/* Program to implement broadcast operation using ELROS
 * USAGE: ecc –o broadcast broadcast.ec
 *              broadcast start_port_number total_number_of_processes process_number
 * e.g;          broadcast 1050 5 1 –creates 5 processes with the root process at port
 *               number 1050, process#2 at port number 1051 and so on.
 */

#include <stdio.h>
#include <string.h>
#include ”interfaces/broadcast.i”
#include ”include_elros.h”

main(argc, argv)
int argc;
char *argv[];
```

greater than zero. If it is, the counter is passed on to the succeeding process. If the counter is equal to zero, all the socket connections are disconnected by calling the *close()* function and the process is terminated.

4.2.3 External Process Code

The external process that provides the initial value of count to process #1 is as follows:

```
/*  External Client that connects to the last process in the ring.*/

#include "include_socket.h"

main(argc, argv)
int argc;
char *argv[];
{
  int sockfd,                          /* socket descriptor */
      server_port;                     /* port number of the server */
  char buff[BUF_SIZ];                  /* send and receive buffers */
  struct sockaddr_in serv_addr;        /* address structure of the server */
  extern void init_addr();             /* function to initialize the address structure */
  extern void error();                 /* function to display the error message and terminate */
  /* Fill in the structure "serv_addr" with the address of the
     server that we want to connect with */
  server_port = atoi(argv[1]);
  init_addr(&serv_addr, server_port);

  /* Open a TCP socket (an Internet stream socket). */
  if((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    error("external_client: Can't open stream socket");
  /* Connect to the server */
    if(connect(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
    error("external_client: Can't connect to server\n");
  /* Send string 20 to the server */
  strcpy(buff, "20");
  send(sockfd, buff, sizeof(buff), 0);
  close(sockfd);
}
```

4.2.4 Description of External Process Code

The external process is spawned by the last (highest–numbered) process in the ring. This process initializes its address structure variable with the address of process #1 (to which this process acts as a client) in the ring. The *socket* function call is then made to allocate a socket. The *connect* function call establishes a connection with process #1. The initial value of the counter is passed to process #1 by calling the *send* function. Finally, connection with process #1 is terminated by calling the *close* function.

5 Collective Operations

Collective operations are operations performed on a group of processes. ELROS does not provide any primitives that involve a group of processes. We have implemented a few commonly used

```
        send(sockfd1, buff, sizeof(buff), 0);
    }
    while(recv(newsockfd, buff, sizeof(buff), 0) > 0)
    {
        count = atoi(buff);
        printf("node%d: count = %d\n", node_num, count);
        count = count – 1;
        if(count >= 0)
        {
            sprintf(buff, "%d", count);
            send(sockfd1, buff, sizeof(buff), 0);
        }
        else
        {
            close(newsockfd);      /* parent process */
            close(sockfd);
            close(sockfd1);
        }
    }
}
```

4.2.2 Description

1. As in the ELROS version, process #1 will spawn process #2, calls *Accept()* twice and blocks waiting for the process external to the ring and the last process in the ring to call *connect()* to establish a connection with process #1.
2. Process #2 will spawn process #3, calls *Accept()* and blocks waiting for process #3 to call *connect()* to establish a connection with process #2.
3. Similarly all other processes (except the highest–numbered process) will spawn their next higher numbered process, calls *Accept()* and blocks waiting for these processes to call *connect()* to establish a connection with the process that spawned them.
4. The last process (having the highest process #) will spawn the external process and calls *connect()* to establish a connection with process #1. Process #1 establishes connection with the last process and the external process, and calls *connect()* to establish a connection connection with process #2.
5. Process #2 get connected to process #1 and calls *connect()* and establishes connection with process #3.
6. Similarly all other processes (except the last, highest–numbered process) calls *connect()* and get connected to the servers in the process preceding these processes.
7. Process #1 get a counter from the external process, decrements the counter and checks if it is greater than zero. If it is, the counter is passed on to process #2. If the counter is equal to zero, all the socket connections are disconnected by calling *close()* and the process is terminated.
8. Similarly each of the other processes receives a counter from its respective preceding process, decrements the counter and checks to see if it is

```
sprintf(next_node, ”%d”, node_num+1);
if(node_num < total_nodes)
{
     if(fork() == 0)
          execl(”./ring”, ”ring”, argv[1], argv[2], next_node, NULL);
}
else
{
     if(fork() == 0)
          execl(”./external”, ”external”, argv[1], NULL);
}
/* Open a TCP socket (an Internet stream socket). */
if((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
     error(”server: Can’t open stream socket\n”);
/* Initialize the address structure for this node */
port = start_port_num + node_num – 1;  /* port number of this node */
init_addr(&address, port);           /* Initialize the address structure */
/* Bind our local address so that the client can send to us. */
if(bind(sockfd, (struct sockaddr *) &address, sizeof(address)) < 0)
     error(”server: Can‘t bind local address\n”);
listen(sockfd, 5);
/*
 * Wait for a connection from a client process.
 * This is an example of a concurrent server.
 */
if(node_num != total_nodes)
{
     newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
if(newsockfd < 0)
          error(”node%d: error accepting the client\n”);
}
if(node_num == 1)
{
     ext_sockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
     if(ext_sockfd < 0)
          error(”node1: error accepting external socket\n”);
}
/* Initialize the addr structure of the server */
if(node_num != total_nodes)
     next_port = port + 1;               /* port number of the server */
else
     next_port = start_port_num;
init_addr(&address, next_port);
/* Open TCP socket */
if((sockfd1 = socket(AF_INET, SOCK_STREAM, 0)) < 0)
     error(”server1: Can’t open stream socket”);
/* Connect to server */
if(connect(sockfd1, (struct sockaddr *) &address, sizeof(address)) < 0)
     error(”client: can’t connect to server”, node_num);
if(node_num == total_nodes)
{
     newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
}
if(node_num == 1)
{
     recv(ext_sockfd, buff, sizeof(buff), 0);
```

```
        give_counter(count);
    }
    Unbind(n1);
}
```

4.1.4 Description

The external process calls *Bind_addr()* as an initiator (client) and wait for process #1 in the ring to call *Bind_addr()* as a responder (server). When process #1 does, a connection is established. The external process calls the *give_counter* function defined in the *ring* interface to send a counter (whose value is twenty) to process #1. Finally the external process calls *Unbind()* to disconnect from process #1. The counter value of twenty was chosen in our example because, with five processes there will be four rounds of message passing across the ring.

4.2 Socket Version

Next we present a C program that uses sockets to create a set of processes connected in the form of a ring. This program uses a reliable, connection–oriented byte streams for passing messages.

4.2.1 Program Code

```
/* Program that uses sockets to create processes connected in the form of ring
 * USAGE:  cc –o ring ring.c
 *         cc –o external external.c
 *         ring start_port_no. total_processes process_no.
 * e.g;    ring 3050 5 1 –will start 5 processes.
 */
 #include ”include_socket.h”

main(argc, argv)
int argc;
char *argv[];
{
   int sockfd,              /* socket descriptor */
     sockfd1,               /* socket connected to the next process */
     newsockfd,             /* socket to receive counter from the previous process */
     clilen,,               /* length of the client's address */
     count,                 /* counter passed around the loop */
     start_port_num,        /* port number of process #1 in the ring */
     total_nodes,           /* total number of nodes in the ring */
     node_num,              /* node number of this process */
     next_port,             /* port number of the next process in the ring */
     ext_sockfd;            /* socket connected to the external process */
     short port,            /* port number of this process */
     ext_port;              /* port number of the external process */
  char buff[BUF_SIZ],       /* buffer used for send and receive operations */
     next_node[NODES];      /* next node in the ring */
  struct sockaddr_in cli_addr,      /* structure to hold the client address */
     address,                       /* structure to hold address of this process */
     ext_address;                   /* struct to hold address of the external process */
   extern void init_addr();         /* function to initialize the address structure */
   extern void error();             /* function to display the error message and terminate the program */
     start_port_num = atoi(argv[1]);
     total_nodes = atoi(argv[2]);
     node_num = atoi(argv[3]);
```

and blocks waiting for the succeeding process to call *Bind_addr()* as a client for these servers.

4. The last process (process #n) will spawn an external process and executes *Bind_address()* as a client for the server in process #1. Process #1 get connected to process #n and the external process and calls *Bind_addr()* as a client to the server in process #2.
5. Process #2 get connected to process #1, executes *Bind_addr()* as a client to the server in process #3.
6. Similarly all the other processes (except the last process) execute *Bind_address()* as a client to the server in the process proceeding it and get connected to it, thus forming a ring structure of processes
7. Process #1 get a counter from the external process, decrements it and checks if it is greater than zero. If it is greater than zero, the counter is passed on to process #2. If the counter is equal to zero, all the connections are disconnected by calling the *Unbind()* function and process is terminated.
8. Similarly each of the other processes receives a counter from its respective preceding process, decrement the counter and checks to see if it is greater than zero. If it is, the counter is passed on to the succeeding process. If it is equal to zero, calls *Unbind()* function to disconnect the connections that were established and get terminates.

4.1.3 External Process Code

The code for the external process mentioned in the previous section is shown below. This process provides initial value of count to process #1 to initiate the ”example code”. It uses the same interface definition used in the ring program.

```
/* external process that provides the initial counter value to the process #1 */
#include ”interfaces/ring.i          /* import ring interface definition */
#include ”include_elros.h”                    /* contains header files and some functions */
main(argc, argv)
int argc;
char *argv[];
{
    int bound;                      /* flag to be set while this process is bound to process #1 */
    char srv_ip_port[IP_PRT_LEN];           /* IP+port number of the server */
    Control(ring) n1;               /* communication instance to bind to process #1 */
    Bind_address addr_1;            /* address structure variable */
    counter count;
    sprintf(srv_ip_port, ”%s%s”, IP_ADDRESS, argv[1]);
    init_puresocket_addr(&addr_1, Initiator, srv_ip_port);
    Bind_addr(n1, Asn_1_ber, &addr_1, TIMEOUT);
    puts(”external_process: After binding to process#1\n”);
     *count = 20;
    bound = TRUE;
    /* pass counter to the first process in the ring */
    Give n1
    {
```

```
after decrements it */
    while(bound)
    {
        Accept previous_node
          {
            On decrease_counter(count)
              {
                printf("process%d: count = %d\n", node_num, *count);
                if(*count == 0)
                {
                    Unbind(next_node);
                    Unbind(previous_node);
                    bound = FALSE;
                }
                decrease_function(count);
                Give next_node
                  {
                    decrease_counter(count);
                }
            }
        }
        /* if accept fails because the previous process has disconnected from
        this process,is connect from all the processes to which this process is  connected */
        Exceptions
          {
            On Done
              {
                Unbind(next_node);
                Unbind(previous_node);
                Unbind(extern_node);
                bound = FALSE;
            }
        }
    }
}
/* function to decrement the counter */
void decrease_function(count)
counter count;
{
    *count = *count – 1;
}
```

4.1.2 Description

The above program works as follows:

1. Process #1 will spawn process #2 , calls *Bind_addr()* twice as a server and blocks waiting for the process external to the ring and the last process in the ring (process #n) to calls *Bind_addr()* as clients for this server.
2. Process #2 will spawn process #3, calls *Bind_addr()* as a server and blocks waiting for process #3 to call *Bind_addr()* as a client for this server.
3. Similarly all other processes (except the highest numbered process) will spawn the next higher numbered process, calls *Bind_addr()* as a server

```
then as a client to the next process. If this is the last process, execute Bind_addr() as client (to
the first process) and and the as a server (to previous process) */
if(node_num != total_nodes)
{
     init_puresocket_addr(&addr,  Responder, ip_port_num);
     Bind_addr(previous_node, Asn_1_ber, &addr, TIMEOUT);
     init_puresocket_addr(&addr,  Initiator, next_node_addr);
     Bind_addr(next_node, Asn_1_ber, &addr, TIMEOUT);
}
else
{
     init_puresocket_addr(&addr,  Initiator, next_node_addr);
     Bind_addr(next_node, Asn_1_ber, &addr, TIMEOUT);
     init_puresocket_addr(&addr,  Responder, ip_port_num);
     Bind_addr(previous_node, Asn_1_ber, &addr, TIMEOUT);
}
/* if this is the first process, establish connection with an
     external node  as a server */
if(node_num == 1)
{
     init_puresocket_addr(&addr,  Responder, ip_port_num);
     Bind_addr(extern_node, Asn_1_ber, &addr, TIMEOUT);
}
/* if this is the first process, accept initial value of the counter from the external node and then disconnect
     form it */
if(node_num == 1)
{
     Accept extern_node
        {
          On give_counter(count)
             {
               printf("process%d: count = %d\n", node_num, *count);
               if(*count == 0)
               {
                    Unbind(next_node);
                    Unbind(previous_node);
                    bound = FALSE;
               }
               decrease_function(count);
               Give next_node
                  {
                    decrease_counter(count);
               }
          }
     }
     Exceptions
        {
          On Done
             {
               Unbind(extern_node);
          }
     }
}
/* initialize the value of bound */
bound = TRUE;
/* while this process is bound to it's previous process, accept counter from it and pass it to the next process
```

it. Process #2 will then pass the counter to process #3 after decreasing it. This way the counter is passed around the ring until its value is zero.

```
/* Program to spawn processes and connected them in the form of a ring
 *  USAGE: ecc –o ring ring.ec
 *              ecc –o external external.ec
 *              ring start_port_no. total_processes process_no.
 *  e.g;       external process code;   ring 3050 5 1 –will start 5 processes.
 */
#include <stdio.h>
#include ”interfaces/ring.i”    /* import the ring interface */
#include ”include_elros.h”
main(argc, argv)
int argc;
char *argv[];
{
     int start_port_num,            /* port number of the first process in the ring */
     node_num,                      /* node number of this process */
     total_nodes;                   /* total number of nodes */
     char next_node_addr[IP_PRT_LEN],/     * IP+port number of the next process in the ring */
     next_node_num[NOD_LEN], /* node number of the next process in the ring */
     ip_port_num[IP_PRT_LEN],/* IP+port number of this node */
     ip_addr[IP_LEN];               * IP address of this node */
     short     bound;               /* true until this process is connected to its previous and  next processes
                                    in the ring */
     Control(ring) next_node        /* communication instance for the next process in the ring */
     previous_node,                 /* communication instance for the previous process in the ring */
     extern_node;                   /* communication instance for the external process in the ring */
     Bind_address    addr;          /* address structure variable */
   counter count;                   /* counter that is passed around the ring */
     void decrease_function();      /* function to decrement counter value */
     start_port_num = atoi(argv[1]);
     node_num = atoi(argv[3]);
     total_nodes = atoi(argv[2]);
     /* calculate the port address of the next process in the ring */
     if(node_num != total_nodes)
          sprintf(next_node_addr,”%s%d”,IP_ADDRESS, start_port_num+node_num);
     else
          sprintf(next_node_addr, ”%s%d”, IP_ADDRESS, start_port_num);
     sprintf(next_node_num, ”%d”, node_num+1);
     /* if this is not the last process in the ring,spawn the next process,
      else spawn the external process that provides the initial counter value      to the first process */
     if(node_num < total_nodes)
     {
          if(fork() == 0)
               execl(”./ring”, ”ring”, argv[1], argv[2], next_node_num, NULL);
     }
     else
     {
          if(fork() == 0)
               execl(”./external”, ”external”, argv[1], NULL);
     }
     /* compute ip_port_number of this process */
     sprintf(ip_port_num, ”%s%i”, IP_ADDRESS,start_port_num+node_num–1);
     /* if this is not the last process in the ring, execute Bind_addr() as a server to the previous process) and
```

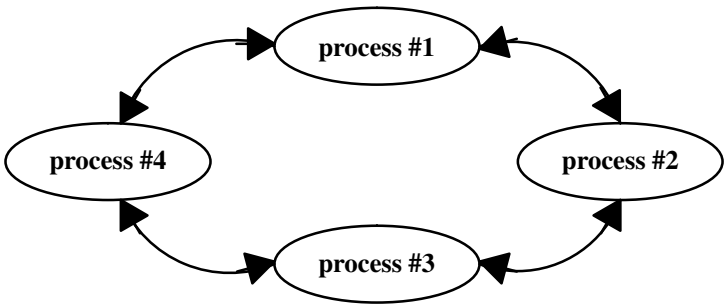

Figure 1 – Processes connected in the form of a ring

4.1 ELROS Version

The interface definition needed to create a set of processes connected in the form of a ring is given below.

```
/* from ring.i */
/* interface definition for ring program */
#define op_num1 10        /* operation number for decrease_counter */
#define op_num2 20        /* operation number for give_counter */
 Interface ring
 Array Integer(1) counter[1];
Invoke   decrease_counter op_num1
         counter;
Invoke   give_counter op_num2
         counter;
End;
```

The name of the interface is "*ring*". The interface contains two operations *decrease_counter()* and *give_counter()*. Both operations have one argument of type counter, which is an integer pointer. Operation *give_counter()* is used by a process external to the ring to pass the initial value of count to the first process (in the ring). Operation *decrease_counter()* is used to pass counter within the ring between processes. The program that uses the above interface definition is shown below. The statement "*#include interface/ring.i*" is used to import the interface stored in the file *interface/ring.i* into this program. Each process (except the last process) acts as a server to the one succeeding it and as a client for the one preceding it. *(e.g.*, process #1 acts as a server to process #4 and as a client to process #2). The last process in the ring acts as a client to the first process and as a server to the one preceding it.

4.1.1 Program Code

The program presented below will spawn the desired number of processes and connects them in the form of a ring with each process connected to two other processes. Process #1 receives a counter from an external process to which it is connected and passes it to process #2 after decreasing

using sockets, the programmer has explicitly to take care of the data types since sockets treat all messages as character strings.

6. ELROS allows the programmer to define more than one operation request by a client to a server. The server first accepts a service request from a client using the *Accept* statement. Next, using the *on* clause, the server determines the name of the operation to which a service request was made and then executes the set of instructions for that operation.
7. Initial connection establishment between the client and the server is simple. Both the client and the server have first to initialize the address structure of the server by calling *init_puresocket_addr()* specifying the role of a initiator (client) or a responder (server). Next the server blocks after calling *Bind_addr().* A connection is established when the client calls *Bind_addr()*.

3.2 Disadvantages of using ELROS

1. The programmer has explicitly to pass IP address of the server to *init_puresocket().* There is no provision to pass the server name instead of its IP address.
2. There is no provision for the client process to test if there is a server process ready to accept connection request from this client. If the client process tries to establish a connection with a server by executing *Bind_addr()*, it will be aborted should there be no server process ready to accept its connection request. This was one of the main problems we encountered while using ELROS. One of the solutions to this problem is to specify an *Exceptions* clause to *Bind_addr()* in the client process and to take remedial measures within this *Exceptions* clause.
3. The instance–variable argument for *Bind_addr()* cannot be an element of an array. That is, we cannot pass an element such as process_instance as an instance variable to this function. In case we have an array of instances, each time an element of this array is to be passed to *Bind_addr()*, we have to copy it to a temporary variable and then pass the temporary variable to *Bind_addr()*. This is evidently a lexical limitation.
4. ELROS does not provide a routine to test if there is any message to be received before actually receiving the message by executing the *Accept* statement (Non–blocking or "probe").
5. ELROS does not work with FORTRAN. Thus it is not possible for the existing FORTRAN programs to use ELROS.

4 Processes connected in the form of a ring

This is a common form of interconnection used in distributed computing. Here each process is connected directly to two other processes to form a ring structure (figure 1).

ELROS supports both synchronous and asynchronous mode of communication between the client and the server. In an synchronous communication, the client process requests service from a server process and blocks until the server accepts the request, processes the data and return the result to the client process. In an asynchronous communication, the client process can continue execution after requesting service from a server process. The *Give* statement is used by the client to asynchronously send one or more operations to the server. The server receives these operations using the *Accept* statement. The *Call* statements can be used for synchronous operations.

Various kinds of errors might result while using ELROS statements. ELROS provides exception handling facilities to take care of these conditions. An *Exceptions* clause can be attached to most of the ELROS statements. Each of these *Exceptions* clause may contain one or more *On* clauses that specify exceptional conditions that may occur during the execution of the ELROS statement to which the *Exceptions* clause is attached. The C statements within the corresponding *On* clause are executed in case of an exception. A default *Exceptions* clause will be executed in case the programmer fails to provide one for a particular ELROS statement.

Debugging tools available in ELROS can be used to debug application programs written using ELROS. These include routines to print variables declared as ELROS type, routines to trace execution of ELROS programs, and utilities for viewing the execution of ELROS statements in real-time [2].

3 Comparison of ELROS and Sockets

Some of the advantages and disadvantages of using ELROS instead of using sockets directly are discussed below [2, 5].

3.1 Advantages of using ELROS

1. It is convenient to use. ELROS statements are embedded in a conventional language like C. The ELROS preprocessor translates these statements to the conventional language statements.
2. Asynchronous interactions between the client and the server can easily be implemented using ELROS.
3. ELROS provides good exception handling capabilities. The programmer can specify actions to be taken for each exception.
4. ELROS provides a set of routines for debugging purposes. The programmer can print the values of variables used in the ELROS programs using the print routines. In addition the programmer can use the logging routines to trace the execution of an ELROS program.
5. ELROS allows programmer to define many different data types including arrays and records in the interface definition. If message passing is done

1 Introduction

ELROS (Embedded Language for Remote Operation Services) is a ‘‘protocol compiler” developed at Lawrence Livermore National Laboratory for programming distributed applications. Detailed tutorial on ELROS is found elsewhere [2, 3]. Developing distributed applications using ELROS is simple since service requests can be embedded in a conventional language such as C. The ELROS preprocessor translates these ELROS statements into the conventional language. Heterogeneity in ELROS is achieved using **ASN.1** encoding and decoding. ELROS provides primitives for both synchronous as well as asynchronous message passing and ELROS can encode/decode complex, even recursive data types. Good exception handling and debugging capabilities supported by ELROS aid in the development of distributed applications.

Although ELROS provides a good programming interface for remote services, it does not provide collective operations, which are valuable for developing distributed scientific applications. This motivated us to implement a few frequently used collective operations (such as broadcast and collapse operations) using ELROS. ELROS Version 2.1 was used to implement these operations. Internally ELROS uses socket system calls to pass messages between processes. We also implemented a few collective operations using sockets for comparison purposes.

In this paper, we present a few of the example programs that we wrote using ELROS. We also present the equivalent code that uses sockets in order to illustrate the ease of programming using ELROS. In our examples, we italicized statements that are specific to ELROS and sockets to improve readability. Programs presented in this paper were tested on Sun workstations. These programs will spawn processes on a single workstation. We preferred this approach because of its simplicity (for demonstration), although we could spawn processes on different workstations with some modification to these programs. Features offered by ELROS compared to those obtainable using sockets system calls directly are discussed [2, 5].

2 ELROS Overview

Application programs written using ELROS contain two main parts: Interface definition statements and executable statements. Interface definition statements contain operation definitions and the data type (ELROS data type) of the parameters to these operations. ELROS provides all of the basic **ASN.1** data types [2]. More than one operation can be defined in an interface definition. Interface definition statements are usually stored in a file separate from that containing executable statements. Interfaces are imported (using *#include* statements) by programs containing ELROS executable statement that uses operations defined in those interfaces.

Collective Operations Using ELROS and Sockets *†

Kishore Viswanathan
Anthony Skjellum

Abstract

ELROS, an acronym for Embedded Language for Remote Operation Service, was developed at Lawerence Livermore National Laboratory for programming distributed applications. ELROS statements can be embedded in a conventional language such as C. This makes it easier to develop distributed applications. ELROS supports both synchronous and asynchronous operations. ELROS also provides a good exception handling facility. Although ELROS provides a good programming interface, it does not provide primitives for collective operations. In this paper we present some of the collective operations that were written using ELROS. We also present programs to demonstrate how these operations can be implemented using sockets. Advantages and disadvantages of using ELROS over sockets are also discussed.

March 7, 1994
Department of Computer Science & NSF Engineering Research Center
Mississippi State University

* This work was supported by the NSF Engineering Research Center for Computational Field Simulation, Mississippi State University.

† Revision 1.0; March 7, 1994.