A System for Resource-Sharing In a Distributed Environment - RIDE

Priscilla M. Lu

Bell Laboratories
Naperville, IL 60540

ABSTRACT

RIDE (Resource-Sharing In a Distributed Environment) has been designed and implemented for the UNIX™ operating system to provide sharing of remote files, remote process invocation and interprocess communication in a distributed computing environment. The system is designed to support a uniform interface for both local and remote access in a network. The user programs can be executed in a single or multiple machine environment without any program modification.

1. INTRODUCTION

There is an increasing need for distributed system support so that programs can operate independently of the resource distribution in the network. A desirable characteristic of a network operating system is to make processor boundaries transparent to the user programs for remote access. A uniform interface for both inter and intraprocess communication provides flexibility in functional distribution within the network and accommodates for system growth by insulating physical hardware configuration from the user programs. It also eliminates some of the complexity for interprocessor considerations in the design of software that executes in a distributed system.

RIDE (Resource-sharing In a Distributed Environment) is a system that aims at supporting the above characteristics in a network. It supports resource sharing in a distributed computing environment under the UNIX™ operating system[1,2], by providing the following facilities:

- remote file access,
- remote process invocation and
- interprocess communication.

These interprocessor activities can be achieved without modifications to the source of the existing programs that are designed and implementd to execute in a single machine. The enhancements to facilitate remote access are introduced in the operating system's input and output routines. These enhancements are hidden from the user's program because the I/O function calls are the same for both local and remote access. Programs which require remote access would have to be reloaded with these enhanced I/O functions. It differs from the traditional approaches in network interfaces in the following ways.

1. RIDE supports a program level interface that gives programmers a greater degree of flexibility in handling remote execution sequences and synchronizations.
2. No new commands have been introduced at the user level for remote system accesses.
3. An executing process can access multiple machine resources simultaneously.

Similar work has been done in network communication for remote file access[3] and interprocess cummunication[4]. The previous work emphasized a master/slave file server, and multi-tasking using n-plex data paths. RIDE provides the UNIX operating system with the abilities to access remote files and communicate or invoke remote processes. This system was designed to replace the network interface programs that previously operated on the SPIDER[5] system.

CH1515-6/79/0000-0427$00.75 © 1979 IEEE

The following sections cover some background information on RIDE, the system overview, user capabilities and a description of the high-level design.

2. BACKGROUND

RIDE uses key concepts in the UNIX operating system. It supports the process-oriented execution environment and provides a convenient means for remote process invocation and file access. It depends on UNIX I/O redirection to achieve these capabilities.

The UNIX standard I/O can be redirected either within a user program or in the command language interface specification. The command programming language, shell6, provides a user interface to the process-related facilities and an environment for the processes to execute in. It facilitates redirection of I/O at the command level. In RIDE, remote I/O is redirected to a logical channel between machines.

RIDE uses the process creation primitives in the UNIX operating system for invoking remote commands. The passing of environment from parent to child process is realized through the system primitive "fork". The system call "exec" can be used for branching control to the process indicated in the parameter. The combination of fork and exec gives users the ability to synchronize activities between parent and child process within a program.

The interprocess communication between machines uses "pipes". A pipe7,8 in the UNIX operating system is an open file used for connecting two interacting processes. The output of one process is directed to the input of another. The pipe in RIDE between processes interacting on different machines is actually a logical channel. The logical channel is implemented through the use of a multiplexor. It assigns and unassigns communication paths between user process and the remote RIDE network process.

There are some limitations to using pipes for communication because processes are blocked as a result of I/O system calls. This prevents the process from handling multiple data paths concurrently. Interprocess communication in RIDE is being enhanced to support real time system application to facilitate this need for multiple machines. These enhancements are compatible to the real-time UNIX operating system9, where non-related processes can communicate through

- messages,
- shared segments,
- signals/event flags.

The multiplexor in RIDE is implemented using messages and shared segments in an enhanced version of the UNIX operating system. RIDE assumes the availability of the lower layers of network communication. These include:

- flow control,
- message switching functions, and
- link interface (I/O driver).

RIDE was initially designed to operate on Datakit10,11, and used the Datakit packet switching and data transmission modules for interfacing with the network. (Refer to Figure 1)

3. SYSTEM OVERIVEW

The system consists of a login supervisor on the remote machine which waits for incoming requests. The Rlibrary (remote I/O library) and the multiplexor on the local machine sets up the connection and assembles the protocol for remote access. The login supervisor on the destination machine creates a file server for each process that requests for remote file access. If the incoming request is a process invocation, or a request for interprocess communication, the login supervisor will establish redirection of input, and output and error between the invoker and the new remote process.

The remote process and file server inherit signals, standard input/output and error from the local environment. Interrupts can be sent to these remote processes from the local machine.

RIDE aims at providing a simple mechanism for the user to access remote files, execute and communicate with remote processes from any remote machine on the network. The UNIX commands have been reloaded with the Rlibrary to provide the necessary capabilities of a network operating system. For example, "diff", which compares the differences between two files can now access files on two different machines and print the result of the differences on standard output. In the example below, the "diff" command compares file1 on UNIXA with file2 on UNIXB and directs the standard output ">" into file3, which resides on the user's machine.

```
diff UNIXA!file1 UNIXB!file2 > file3
```

When the file name is not preceded by a machine name, the file is assumed to be on the local machine. The explicit qualification of machine names may be eliminated if users provide a network directory of file names and program names to the network configuration table supported by RIDE.

RIDE supports both program, and command level access to remote machines. The command level access is provided through reloading existing UNIX commands with the Rlibrary to obtain remote access.

The system has the following characteristics. It is

- I/O driven and uses I/O redirection for access to the remote systems,
- upward compatible with the UNIX operating system, and user programs that use the UNIX I/O system calls,
- flexible to user program modifications and operating system enhancements,
- supports communication to multiple machines in the network within a single executing process (i.e., one local process can simultaneously access resources on several machines on the network).

4. USER CAPABILITIES

RIDE is designed to provide the user a view of the network as a system with resources readily available to the user's program. All files in the network can be accessed as if they resided on the local machine, and processes distributed in the network can be invoked with the existing operating system compatible interfaces.

The emphasis in RIDE is to provide users "interactive" services in the network. This means on-line response to the end-user or executing process for activities across the machines.

4.1 File Access

Remote file access is achieved through the use of enhanced I/O functions in the Rlibrary. The multiplexor handles message switching and network initialization and termination procedures.

A user program that uses UNIX I/O system calls can obtain remote file access by loading the program with the Rlibrary. The Rlibrary handles both remote and local file access using uniform function calls. If the request was for remote access, the Rlibrary will communicate to the login supervisor the request and the return values of the function call will be returned to the executing process. If the access is to a local file, the Rlibrary will call the local I/O routines.

The first access to a remote machine by a process triggers a login procedure. This login procedure is transparent to the user process. The login supervisor responds to this on the remote machine by setting the directory of the remote machine to the user's default directory, and creating a file server dedicated to the communication of the local process. This assumes uniform login identification names on all machines in the network. If the login identification name of the executing process does not exist on the remote machine, a system default directory may be used as the login directory.

The established communication path between the local process and its corresponding file server allows the process to continue subsequent remote file accesses without the overhead of a login procedure. The local process can execute any I/O functions on the remote machine, such as create, link, fstat, etc. The same error codes are used for remote file access. Some new error return codes have been added to indicate network failures.

The file access capability conveniently supports the redirection of input or output of an executing process at the command level. Programs requiring I/O from a remote file do not need any modification to access remote systems.

For example, 'sort < UNIXA!file'.

Here the program "sort" accepts input "<" from a file on machine UNIXA.

Appendix-A shows a simple example of program-level remote file access.

4.2 Process Invocation

Users or programs can invoke a process on a remote machine and the output will be directed to the local machine. This remote job execution allows users to execute any commands on a remote system and treat the output as if the job executed locally.

If a user wishes to find out who is on a remote machine, the UNIX command "who" will list all active users. By preceding the command with the destination machine

name, "UNIXA!who", the command will be executed on the other machine, and the output directed to the user's local standard output. A transparent login procedure uses the user's default directory if the invoker's login identification exists on the remote machine, otherwise it uses the system's default public directory.

The login supervisor handles command level process invocation by executing the following. It

- changes directory to invoker's directory if the login id exists on the remote machine,
- sets standard input, output and error to an I/O handler which handles the protocol for cummunication over the link, and
- creates the requested process.

When the invoked process terminates, control is passed back to the login supervisor.

The program level remote process invocation is handled by the file server on the destination machine. It interprets the request and invokes the process.

4.3 Interprocess Communication

Processes on the UNIX operating system can achieve interprocess communicaton through "pipes". Each process inherits from the shell's standard input and output. If the process resides on a remote machine, the input/output is handled by RIDE and requested over the network.

For example, 'UNIXA!who | grep "brown"'.

The pipe "|" is used for communication between the program "who", which lists all active users on UNIXA, and the local executing program "grep"' which searches for the user id "brown". The output of "who" has been redirected to the local machine and used as standard input to "grep".

The redirection of the input and output on the local machine is handled by enhancements to the shell. The modified shell requests for remote access, and the login supervisor on the remote machine spawns the requested process and directs the input and output of the process to the local machine. This results in a "pipe" realized through actual links between the machines.

Work is continuing to enhance RIDE so that it can provide interprocess communication between non-related processes (i.e., processes with no common parent). The exchange of information between processes on different machines through messages labelled with unique port or process identification would facilitate program design and implementation that could be independent of system network configuration and functional distribution.

5. HIGH-LEVEL SYSTEM DESIGN

High-level protocols reflecting the I/O requests are the basis of communication between the file server and the Rlibrary for file access and process invocation. A set of protocols are used for communication between the enhanced shell and the login supervisor for command invocation. The system consists of 4 major components, the modified remote I/O functions in the Rlibrary, the multiplexor, the login supervisor, and the file server. Figure 2 shows the modules of the system on the local and the remote computer.

5.1 The Rlibrary

The Rlibrary is a set of remote standard I/O routines (e.g., open, close, read, write, lseek, stat) that access remote or local files, and execute remote processes. When the user program calls one of these routines in the Rlibrary, it will determine whether the operation is to be executed on a local or remote computer. If it is a local operation, the regular standard I/O routine is called, otherwise a message is assembled using the function and the parameters of the function. The message is passed to the interface module in the multiplexor for sending to the specified destination machine.

When a child process is created by the local executing process, the remote open files are inherited as part of the environment of the parent process.

5.2 Network Communication

The Network Communication Modules consists of: Encode/Decode Module, Interface Module, the Network Processing Module, and the message multiplexor.

The Encode/Decode Module assembles and disassembles the messages for the Rlibrary and the file server. It performs some of the machine dependent conversions necessary to achieve compatibility between different machines using the UNIX operating system.

The Message Multiplexor lables each incoming and outgoing message with the process and machine identification. It uses the Interface Module for sending and receiving data over the link.

The Interface Module sends and receives messages between the remote processes and the local shell or Rlibrary routines. It performs the following functions:

- identifies destination machine and establishes a logical connection,
- determines the communication status of each channel,
- executes initialization and termination procedures,
- sends and receives protocols,
- performs system control message interpretation,
- detects system transaction failures (i.e., interrupts, login failures, etc.)

The Network Processing Module provides routing information to the multiplexor. It contains:

1. Network Status Table
   - machine to channel mapping,
   - status of channel (available or in use).
2. Process Dependent Information
   - process to machine connection status,
   - file to machine mapping,
   - local to remote file description number, and
   - record of most recent error encounteredwith the machine identification to locate source of error.

5.3 Login Supervisor Module

The login supervisor module runs continuously on each computer in the network. It waits on an open channel for any incoming messages (Refer Figure 3). When the incoming data is a login message, it will execute the login sequence for the user. It sets the directory to the user's default directory. Depending on the incoming request; it either creates a file server for file access, or a user process for remote process execution. The login procedure is necessary only at the first transaction between the local process and the remote machine. Since the Login Module redirects standard input and standard output for communication with the local process, any process invoked by the file server will inherit the same standard input and standard output.

5.4 File Server

A user process performs remote I/O through the Rlibrary routines to a remote file server. There exists one file server for every remote machine that a local process is interacting with.

Error Handling

When the file server detects an error in executing a command, it transmits the error number back to the local program. File server keeps a record of the most recent error, and if the user requests for the textual description of the error, it will transmit the actual error message.

Process Invocation Within Programs

When a request is made to invoke a remote process within a program, the file server will fork and "exec" the process. This new process inherits the file server's standard input and output.

6. STATUS

There are a number of issues that require further study.

1. The problem of handling system protection and security in the network (passwords). The system, however, is designed to accommodate the implementation of any methods adopted for security in the network. The logic for enforcing security can be placed in the login supervisor and the message multiplexor which performs the connection procedure.
2. The passing of environment (such as open files) across machines need further exploration. If a child process is created on a remote machine, the parent's environment on the local machine is not completely inherited.

There are a few limitations to the system. The system does not support simultaneous access by both the parent and the child process.

The enhancements to the I/O routines increased the process size of the user programs. This can be a limitation to large programs.

Since memory is not shared across processors, only data and messages can be exchanged in the network.

Experiments with RIDE were conducted on the Datakit network and a number of point-to-point communication installations. It is a prototype for a network operating system currently under development.

7. SUMMARY AND CONCLUSIONS

RIDE is a means of providing interprocessor communication so that there is a uniform interface for both local and remote access. The destination machines could be transparent to the users if the internal file system supported network information. Similarly a process configuration table could be used to locate process distribution. The system can be used to assist in load balancing. Since the system facilitates remote execution of programs and remote file access, one can develop a method for distributing selected jobs to be executed on larger capacity computers. This may be done to offload an overloaded machine, or to distribute large tasks among machines specially equipped to handle heavy computation. RIDE is a step towards providing this facility in a way that is transparent to the user.

RIDE can be used to access and update distributed data bases within a network. A local process could invoke a remote database management system (DBMS) with a query, and the DBMS can direct the response to the local executing process.

Work is continuing, to extend the interactive capabilities of RIDE to real-time system applications. This would allow programs to use messages, shared segments, event flags and semaphores for interprocessor communication. The advantage of RIDE is in the uniformity of user interface for both local and remote access. The software architecture of distributed system can support a flexible functional distribution across processors using this facility.

ACKNOWLEDGEMENTS

The author thanks A. G. Fraser and G. L. Chesson, who inspired and influenced much of the design of the system and S. Roy and J. M. Scanlon for their comments on this document. The author wishes to acknowledge the efforts of J. Q. Arnold who modified and extended the system to include message multiplexing.

APPENDIX A

The example below shows how a C program[12] can create a file on a machine "UNIXA", open a file on "UNIXB" and copy the data from the second to the first file. The file names can be substituted as a string variable that gets initiated as a parameter to the procedure call.

```
     main() {
:
:
fd0 = creat("UNIXA!file",mode);
if (fd0 < 0){
printf("cannot  creat file on UNIXA");
          exit();
}
fd1 = fopen("UNIXB!file1",2);
if ( fd1 < 0) {
          error("failed on open ");
          exit();
}
nread = read(fd1,rdbuf,nbytes);
nwrite = write(fd0,rdbuf,nread);
:
:
     }
```

REFERENCES

[1] D. M. Ritchie, K. Thompson, "The UNIX Time-Sharing System" The Bell System Technical Journal, July-August 1978, Vol. 57, No. 6, part 2.

[2] D. M. Ritchie, K. Thompson "The UNIX Time Sharing System" Communication of ACM, July 1974, Vol 17, No. 7.

[3] S. F. Holmgren, "Resource Sharing UNIX" COMPCON 78 Proceedings - Computer Communications Network, pp. 302-305.

[4] R. Balocca, "Networking and the Process Structure of UNIX: A Case Study", COMPCON 78 Proceedings - Computer Communications Network, pp. 306-311

[5] A. G. Fraser, "A Virtual Channel Network" Datamation pp. 51-56, February 1975.

[6] S. R. Bourne, "The UNIX Shell" The Bell System Technical Journal, July-August 1978 Vol. 57, No. 6, part 2.

[7] K. Thompson, "UNIX Implementation" The Bell System Technical Journal, July-August 1978 Vol. 57, No. 6,

part 2.

[8] B. W. Kernighan, J. R. Mashey "The UNIX Programming Environment" Software Practice and Experience, Vol. 9, 1-15 (1979).

[9] H. Lycklama, D. L. Bayer, "The MERT Operating System" The Bell System Technical Journal July-August 1978, Vol. 57, No. 6, part 2.

[10] G. L. Chesson, "Datakit Software Arthitecture" ICC Conference Proceedings, 1979.

[11] A. G. Fraser, "Datakit-A Modular Network For Synchronous and Asynchronous Traffic" ICC Conference Proceedings 1979.

[12] D. M. Ritchie, S. C. Johnson, M. E. Lesk, B. W. Kernighan, "The C Programming Language", The Bell System Technical Journal July-August 1978, Vol. 57, No. 6, part 2.

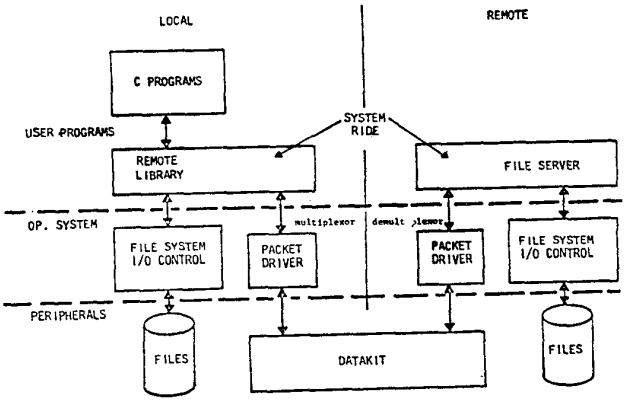

FIGURE 1: RIDE WITH UNIX OPERATING SYSTEM ON DATAKIT

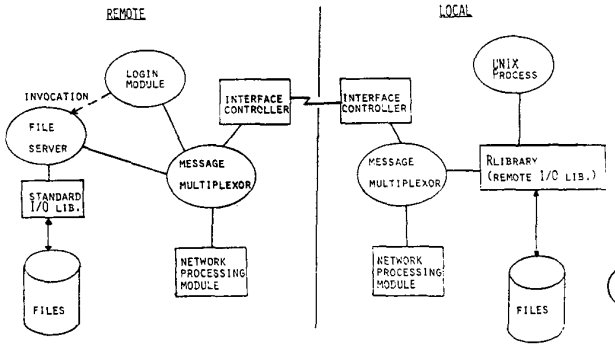

FIGURE 2: RIDE SYSTEM MODULES

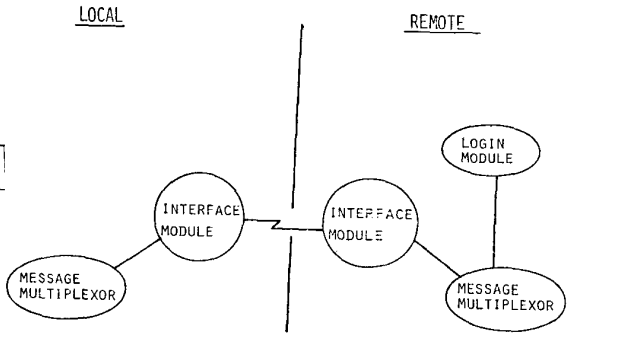

FIGURE 3 : NETWORK PROCESSES IN RIDE(WHEN SYSTEM IS IDLE)