Experience with PARPC [†]

Bruce Martin [††]
Charles Bergan
Walter Burkhard
Jehan-François Pâris [†††]

Computer Systems Research Group
Department of Computer Science and Engineering
University of California, San Diego
La Jolla, California 92093

ABSTRACT

PARPC provides an interprocess communication mechanism based on the semantics of a procedure call. PARPC programs always execute a single logical thread of control but may execute multiple physical threads of control. PARPC provides users with a well defined, high level network process model of execution and a familiar program development model supporting heterogeneous, non-uniform environments. The administrative overhead of PARPC is minimal because users administer their own distributed programs and existing UNIX mechanisms for access control and resource accounting are utilized. Our experiences indicate that PARPC has been an effective system for the development and administration of distributed programs.

1. Introduction

Many distributed algorithms require the capability of sending a message to a set of destinations and getting answers back from some or all of them. Examples of theses algorithms are replicated data consistency protocols, such as majority consensus voting [Giff79] and available copies protocol [Bern83], load balancing algorithms, commit and locking protocols, parallel searches through multiple databases and many distributed software maintenance tasks. The remote procedure call [BiNe84] does not lend itself well to express these semantics as it can only model interactions between a single client and a single server [TaRe85].

The *parallel procedure call* was developed to overcome this limitation [MaBeRu87, Saty86]. A parallel procedure call allows a client process to request the parallel execution of the same procedure in *n* different address spaces in parallel.

We present in this paper our experience in designing and using the PARPC system [MaBeRu87], a parallel remote procedure call system developed at the University of California, San Diego. The PARPC system came about as a result of the development of the Gemini file system testbed [BuMaPa87]. Gemini was built for experimenting with protocols maintaining the consistency of replicated files. While writing the first version of Gemini, we found that producing code required for communication between machines dominated our development time. The implementation of remote connections, authentication, remote processes initiation and parameter

† This work was sponsored in part by grants from the U.C. MICRO Program and NCR Corporation.

†† Author's current address: Hewlett-Packard Laboratories, 1501 Page Mill Road, Palo Alto, California 94304.

††† Author's current address: Department of Computer Science, University of Houston, 4800 Calhoun Road, Houston, TX 77204-3475.

transmission constituted the bulk of the programming effort and hindered us in our efforts to experiment with distributed algorithms.

We decided to develop the PARPC system to directly support parallel procedure calls and a high level model of distributed computation. Furthermore, we decided the system should support program development, execution and administration in the complex distributed computing environment we were experiencing in the Department of Computer Science and Engineering at UCSD.

Our distributed computing environment is complex because it consists of heterogeneous architectures, operating systems, communication protocols and non-uniform file systems. The environment includes a VAX 11/780, a Pyramid 90X, dozens of ATT 3b2 systems, an ATT 3b15, a Celerity 1260D, a CCI 6/32, several subnets of SUN workstations and an NCR Tower. While all of these machines are physically connected through several ethernets, developing and maintaining a distributed program in such a complex environment is tedious at best. A construct like the V kernel process group, which supports multicasting in an homogeneous environment [ChZw85], would not apply.

Throughout this paper we discuss how PARPC bridges complex distributed computing environments. We will refer to a network as *homogeneous* if it contains a single machine architecture *and* a single operating system. Otherwise, it is *heterogeneous*. We classify a file name space as *uniform* if files are globally available through file system calls *and* named the same from all sites in the network.

Our overall distributed computing environment is definitely heterogeneous and non-uniform. However, within this environment, we have homogeneous, uniform subnets (e.g. SUN 3 workstations with uniformly mounted NFS), heterogeneous and uniform subnets (e.g. VAX and SUN systems running NFS) and homogeneous, non-uniform subnets (e.g. ATT 3b2s with local file systems). Thus, we wanted PARPC to gracefully exist in these subenvironments as well. Since the heterogeneous, non-uniform case is the most general environment, the mechanisms that PARPC provides to support it automatically accommodate the other environments.

PARPC supports heterogeneous, non-uniform environments by extending the simple, well understood UNIX model for local program development and execution and system administration. Because the system uses existing UNIX mechanisms to extend the model, our experience has been that distributed program development and system administration is easy.

1.1. Status of the PARPC System

PARPC has been ported and PARPC programs execute on most of the systems in our complex environment. We have ported PARPC to the VAX 11/780 running 4.3 BSD, the Pyramid running OSx, the Sun 3 series running SunOS, the Celerity 1260D running 4.3 BSD, AT&T 3b2 systems running System V and HP Series 300 systems running HP-UX. In addition, PARPC has been distributed to approximately fifteen research and educational institutions.

The implementation of a wide range of distributed programs using PARPC demonstrates the ease of program development and the applicability of the parallel remote procedure call abstraction. We have used PARPC to implement the nested object scheduler described in [Martin87] and [Martin88]. PARPC was used to implement the replicated block server described in [CaLoPa87]. Moreover, the programming and administration tools provided with the system are themselves PARPC programs. These tools are described below.

We now describe our experiences with the PARPC system. In section two we review related work. Section three describes the PARPC programmer model and shows how a typical program would be built using PARPC. Section four describes the administrative requirements of the PARPC system, and section five discusses performance of the PARPC system. More technical information regarding PARPC can be found in [Martin86] and [MaBeRu87].

2. Related Work

Parallel procedure call is similar to replicated procedure call. [Coop85] Since parallel procedure call provides the calling code with control over processing the results, it is a more general construct for expressing distributed algorithms. Traditional remote procedure call was implemented in the Xerox Cedar environment by Birrell and Nelson. [BiNe84] Since then, several remote procedure call systems have been implemented for UNIX including Courier [Coop83] and Sun RPC. [Sun 84b] Remote procedure call provides a model of computation that is limited to both a single *logical* and *physical* thread of control. It is an inadequate model for expressing many distributed algorithms, such as those previously mentioned.

As a remote procedure call system, PARPC is unique in three ways. First, PARPC programs execute a single logical thread of control but may execute multiple physical threads of control. PARPC programs *appear* to be sequential. Secondly, PARPC provides users with a well-defined, high level network process model of execution and a familiar program development model. Finally, the system supports program development in heterogeneous, non-uniform environments.

3. A Programmer's View of PARPC

We have found that fairly unsophisticated C programmers have been able to quickly develop distributed programs using PARPC. We believe this is due to the simplicity of the parallel procedure call construct, the high level model of computation presented to the programmer and the set of UNIX-like development tools provided with PARPC. We discuss each in turn.

3.1. A Parallel Procedure Call

A parallel procedure call is a structured programming construct that is easy for a programmer to understand. Although there may be several *physical* threads of control in a program making parallel procedure calls, there is always a single *logical* thread of control.

A parallel procedure call executes a procedure in *n* different address spaces in parallel. The calling code remains blocked while the *n* procedures execute. The order in which the *n* procedures complete is nondeterministic. As each procedure result becomes available, the caller is unblocked to execute a statement to process the result of the returned call. After executing the statement, the caller reblocks to wait for the next result. This continues until the caller breaks out of the parallel procedure call or until no further results are available.

A parallel procedure, *parproc()*, is invoked from a C program as follows:

`parproc (`*hl, parameters*...) *result statement;*

The first parameter, *hl*, identifies the set of hosts where the procedure is to be executed. PARPC programs manipulate host names as human readable strings.[1] The *result statement* may be a compound statement and may contain `continue` or `break` statements. However, if a `break` is executed, all unprocessed results from the parallel procedures become unavailable.

All data are passed and returned via the parameters. Parameters are designated as either in or out parameters and have copy in and copy out semantics. Since a procedure with no out parameters cannot return data, the calling code cannot depend on results of the remote procedure. In such a case, the calling code does not block and *result statement* is never executed; the parallel remote procedure call is effectively a multicast. [MeBo76]

PARPC programs may have multiple physical threads of control only if there are no logical dependencies between the threads. In particular, procedures executing in parallel cannot communicate with each other. Furthermore, the calling code can proceed in parallel with the parallel procedures only if it does not depend on any of their results. These semantics make PARPC

[1] The system provides a library of operations to manipulate lists of hosts. PARPC provides no mechanism for transparently deciding which hosts to use. Such mechanisms could easily be added on top of PARPC. The example in figure 4 assumes the hosts come from the user.

programs appear sequential, allowing parallel execution in a very controlled fashion.

The parallel procedure call construct is the only mechanism in which a programmer must consider the distributed nature of the program. PARPC programmers are never required to be concerned with connection and authentication protocols, scheduling, data conversion and remote process initiation. These tasks are appropriately handled and hidden by system software. PARPC accomplishes this by providing programmers with a uniform, high-level computation model called a *network process*.

3.2. The Network Process Model of Computation

A network process is a distributed tree of local processes (Figure 1). The root, called the *client*, initiates the computation. Internal nodes, called *servers*, execute procedures and return results. Each node, being a local process, has state. However, local state may only be communicated between nodes via parallel procedure calls. All communication is done between levels; servers cannot communicate among themselves. However, servers may themselves be clients of other servers.

Network processes retain local process semantics. UNIX process operations, such as *fork(), execve()* and *exit()* are analogously defined for the network process. Network process integrity is assured by the system. As with local processes, access to resources is controlled for network processes.

If a machine fails in the network process, the subtree rooted at the failed machine terminates. The code making the parallel procedure call can detect the failure in the result statement but need not be concerned with orphaned servers. Similarly, if the client fails, servers terminate automatically.

PARPC supports two server models: *user* servers and *resident* servers. User servers are simpler to use but do not provide the flexibility of the resident server. The user server model allows ordinary, unprivileged users to develop and execute network processes. User server code is automatically initiated by the PARPC system. Network process semantics and integrity are ensured even though the user has no special privileges. The resident server model, on the other hand, allows privileged resident applications, such as file servers, to be easily constructed. Server nodes of resident applications are not automatically initiated by the PARPC system and are given control over the authentication process. The type of server used is a link-time decision and has no impact on the application code. The PARPC server used to invoke users servers is itself implemented as a resident server.

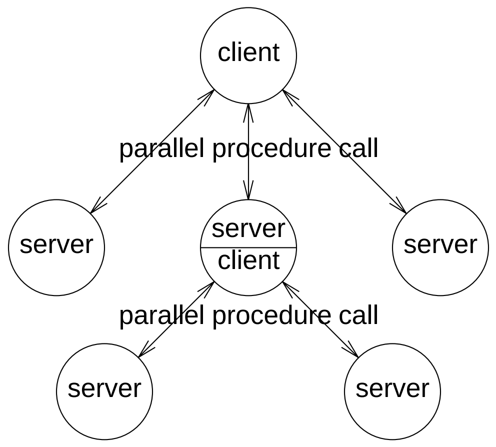

Figure 1: Example Network Process

3.3. PARPC Program Development

PARPC program development easily integrates into the UNIX environment. Building a PARPC program consists of the familiar and simple steps of building a UNIX program. PARPC programs can be built incrementally. Like UNIX programs, PARPC programs are bound together statically. Global consistency is achieved using existing UNIX tools, such as *make* [Feld79] and *lint* [John79]. New tools provided with the system have familiar UNIX-like interfaces.

Many remote procedure call mechanisms, such as Sun RPC [Sun84a] and the Unix implementation of Courier [Coop83], utilize a distinct language to represent externalized data. This forces a program to manipulate both data representations and a programmer to specify a program in more than one language. PARPC programmers specify the interface between clients and servers using a subset of C or C++[KeRi78][Stro86]. The interface is given in a header file as a set of type declarations, procedure declarations and procedure argument specifications. Since the syntax of the interface is designed to be compatible with C and C++ syntax, the same specification file can be used by the compiler or lint to check types of parameter lists of procedure calls and definitions.

Header files containing data type and procedure declarations are processed by a program called *cstub*. *Cstub* produces object code (called *stubs)* that converts data between local and network representations and calls routines in the PARPC run time libraries. Stubs are linked with both the application code and the run time libraries to produce the desired PARPC program.

Cstub statically binds client code to server code by storing a network wide unique identifier, called the *cookie*, in both the client and server stubs. To ensure consistency of client and server code, the cookie is dynamically verified when a node of the network process is created.

The C source files produced by *cstub* are are automatically compiled to produce the following three files:

`client.o,`	an object module containing the client stubs,
`server.o,`	an object module containing the server stubs,
`defs.h,`	a header file containing redefinitions of the parallel procedure calls.

PARPC programs are created in three simple steps. *Cstub* transforms interface specifications into client and server stubs. Application code is compiled. Finally, each node of the PARPC program is created by linking application code, the stubs and the run time libraries together.

For environments where all hosts share a uniform file name space and where each host has the same machine architecture and operating system, *cstub* and the C compiler are sufficient for building PARPC programs. All source code resides in a common location and there is just one copy of all executable programs. In heterogeneous, non-uniform environments, the executable code must reside at each machine. *Cstub* and the C compiler must be executed on each machine in the system. To facilitate this, PARPC provides the replicated compilers, *rcstub* and *rcc.* These tools are themselves implemented as PARPC programs and execute as network processes. Through the use of these tools, source code can be stored in a single location. These tools provide a clean and easy-to-use environment for PARPC program development and maintenance. Figure 2 shows the development steps for four possible distributed environments.

	Uniform File Name Space	**Non-uniform File Name Space**

Homogeneous Architecture and OS	`cstub intf.h`
	`cc client`
	`cc server`
	`mkserver server`
		`rcstub <machines> intf.h`
		`rcc <machines> client`
		`rcc <machines> server`

Heterogeneous Architecture or OS	`rcstub <machines> intf.h`
	`rcc <machines> client`
	`rcc <machines> server`
		`rcstub <machines> intf.h`
		`rcc <machines> client`
		`rcc <machines> server`

Figure 2: PARPC Development Support.

3.4. An Example

We demonstrate the simplicity of the tools with a non-trivial example. We demonstrate the construction of a distributed program to balance the load of a set of printers. The client portion of the program first queries a set of hosts for the status of their respective printers. The client sends the data to the least busiest printer.

Figure 3 gives `printservops.h`, the interface between the client and the printer servers.

```
typedef struct {
     char *data;
     int data_length;
} string;

remote getprinterload (/* hostlist, out int *load */);

remote printfile (/* hostlist, in string filename, out int *error */);
```

Figure 3: `printservops.h`

This file is input to *cstub*. The interface language is a subset of C that is extended with the keywords `remote,` `in` and `out.` Cstub generates client and server stubs for the procedures prefixed by remote. When the C compiler parses the header file, `remote,` `in` and `out` are hidden by the C preprocessor. Other than the `in` and `out` keywords, parameters can be specified using either C++ or ANSI C syntax. Thus, compilers that support type checking of parameters ensure consistency between clients and servers. However, in order to support older C compilers that do not allow parameter specifications, the interface language allows parameters to be specified between C comment symbols `/*` and `*/`.

Figure 4 gives `print.c`, the client algorithm. After building a list of hosts from the command line, the parallel procedure, `getprinterload()`, is executed to find a lightly used printer. As each procedure result becomes available, the statement following the call to `getprinterload()` is evaluated using the value of `load` set by the remote procedure. If no failures occurred and an unused printer was found, the `addfrom` operation sets `printhl` to reference the unused printer. The client then breaks out of the parallel procedure call. The PARPC system discards all further responses about other printers. Finally, if a printer was indeed found, the file is printed by executing `printfile()` on the host referenced by `printhl`. Such a parallel procedure call made to a single host is a traditional remote procedure call.

Figure 5 gives `printserv.c`, the server algorithm.

We now demonstrate how to construct this distributed program in a homogeneous, uniform environment. An example of this environment is a network of architecturally compatible SUN workstations with uniformly mounted NFS. These steps could easily be integrated into a *make* file.

First the client and server stubs, `client.o` and `server.o`, are created as follows:

```
cstub printservops.h
```

Cstub automatically places a network wide unique tag, called the *cookie*, in both `client.o` and `server.o` to ensure consistency between clients and servers. Now the application files are compiled.

```
cc -c print.c printserv.c
```

Next the client executable program is created. The `-l` option tells the loader to use the PARPC client runtime library.

```
cc -o print print.o client.o -lclient
```

Finally, the server executable program is created. The `-l` option tells the loader to use the

```
#include <par_rpc.h>
#include <defs.h>
#include printservops.h

main(argc,argv)
{
    hostlist queryhl, printhl;
    int load, error, minload = MAXINT;
    /* Parse arguments and build host list for querying printer loads */
        :
        :
    /* Parallel procedure call to get printer load from each
       printer given in queryhl.  The expression evaluated for
       each response finds the least loaded printer and sets
       printhl to reference that printer.
    */
    getprinterload(queryhl,&load)
        if (!host_error(queryhl)) { /* if no system error */
            if (load == 0) {  /* found an unused printer, use it. */
                clearhl(printhl);
                addfrom(printhl,queryhl);
                break;
            }
            if (load < minload) {  /* found a less used printer, consider it */
                clearhl(printhl);
                addfrom(printhl,queryhl);
                minload=load;
            }
        }

    if (emptyhl(printhl)) {
        printf("No printers available\n");
        exit(1);
    }

    /* Remote procedure call to send file name to printer
       referenced by printhl. Host_error() indicates whether a system
       failure occurred;  error returns an application error code.
    */
    printfile(printhl,argv[1],&error)
        if (host_error(printhl) || error!=0)
            printf("Error %d printing %s",error,argv[1]);
}
```

Figure 4: `print.c`

PARPC user server runtime library.

```
cc -o printserv printserv.o server.o -lserver
mkserver printserv
```

Files containing executable server nodes are automatically invoked by the parent PARPC server. The `mkserver` program lets the system know about the new server by creating links to the server file. The link is named by the directory where the parent PARPC server executes and the unique cookie. Users own and control access to server files using standard UNIX file access modes. Since client nodes identify server nodes indirectly via the cookie, users may rename files containing server nodes without problems.

Next we demonstrate how to construct this same distributed program in a heterogeneous, non-uniform environment. We will construct the program for a Pyramid system, called *beowulf*, for an ATT 3B2 system, called *ishtar* and for a subnetwork of SUN workstations. The SUN workstations form a homogeneous and uniform subnet; therefore, we need only construct it using a single machine, called *napoli*. The PARPC program will be available from all of the SUN workstations in

```
#include <par_rpc.h>
#include printservops.h

remote getprinterload (qhl, load)
    hostlist qhl;
    int *load;
{
    /* return current load for local printer */
}

remote printfile (phl, filename, error)
    string filename;
    int *error;
{
    /* print file named by filename on local printer */
    /* set error if trouble occurred printing the file */
}
```

Figure 5: `printserv.c`

the subnet.

First the client and server stubs are created around the network as follows:

```
rcstub beowulf:/usr/print/src ishtar:/local/src napoli: printservops.h
```

Rcstub invokes *cstub* on hosts beowulf, ishtar and napoli. *Cstub* creates `client.o` and `server.o` in directory `/usr/print/src` on beowulf, in `/local/src` on ishtar and in the current directory on napoli. Each host is sent a copy of `printservops.h`, as well as any non-system *include* files. To accommodate differences in heterogeneous environments, host-specific *cstub* options may be specified on the command line.

Rcstub automatically generates a unique network-wide cookie and sends it as a argument to each remote invocation of *cstub*. The cookie globally defines the interface around the network.

Next the application code is compiled around the network as follows:

```
rcc -c beowulf:/usr/print/src ishtar:/local/src napoli: print.c printserv.c
```

Rcc invokes the C compiler around the network. As with *rcstub*, *rcc* sends all necessary source files to each host where the compilation is to take place. Arguments to *rcc* are a list of hosts, host-specific *cc* options and default *cc* options that are applied to all hosts. *Rcc* may be used to build non-distributed UNIX programs as well.

With the appropriate options, *rcc* may be used to invoke the linker around the network to produce executable PARPC programs. Thus, the client and server executable files are created around the network as follows:

```
rcc -o print beowulf:/usr/print/src ishtar:/local/src napoli: print.o client.o -lclient
rcc -o printserv beowulf:/usr/print/src ishtar:/local/src napoli: printserv.o server.o -lserver
```

To inform the PARPC system about the existence of a new server, `rcc` automatically invokes the `mkserver` program around the network.

4. An Administrator's View of PARPC

A remote procedure call package affects two areas of system administration: security and system resources. It must ensure that a remote user cannot gain access to any resources to which he would normally be denied, and that any resources used by a remote process are charged to him. UNIX provides simple and elegant mechanisms for both access control and accounting in a local environment. Our goal was to extend the use of these mechanisms in a heterogeneous, non-uniform environment. We designed a simple solution. PARPC uses a translation table which ensures all processes created by a remote user are run as a local equivalent. This allows the standard UNIX accounting and access control facilities to be used. By supporting the use of standard UNIX facilities on PARPC programs, the additional administrative overhead of PARPC is minimal.

4.1. Access control

Access control mechanisms restrict the access capabilities of remote processes. PARPC extends the UNIX access control mechanisms by propagating user and group identifiers across hosts. This poses a problem in a heterogeneous network environment since user identifiers are not necessarily consistent across machines. PARPC provides a translation mechanism which, given a host and a user identifier, returns the local representative of the user. These equivalencies are given in an installation-supplied network equivalency file which lists equivalent users on different hosts. For example, on our local computer network, this equivalency file contains the line:

```
424@napoli 424@roma 424@thor 417@ishtar 419@beowulf
```

A network process whose client was created by a user whose (effective) uid is 417 on ishtar is given (effective) uid 424 on napoli. PARPC also supports the idea of a *network group* . A network group identifier is network-wide unique integer representing a group existing on all machines. PARPC allows the administrator to specify a global constant called the *minimum group id* . Any groups to which the calling procedure belongs which are above this constant are propagated to the remote process. This allows for the creation of network wide groups.

This translation scheme requires the secure transmission of the caller's credentials. We have implemented this secure transmission through the use of a program called *start* . *Start* is responsible for ascertaining the user id, effective user id, group list (BSD) and effective group id (System V) of the caller and transmitting them to the server. Start itself is a privileged UNIX program and transmits this information to the host via a root only socket. By using a root only socket, we ensure that no user process can transmit a false identity to the remote host. The start program tells the remote process which port the caller will be using. This enables the remote process to verify that no user can surreptitiously break into a communication and gain the access permissions of the caller.

Since the UNIX access control mechanism is utilized, clients or servers may be setuid programs. This allows PARPC programs to be executed by users for which no local equivalent exists on remote hosts.

4.2. Server Files

To initiate a remote service, the client must somehow identify the server code he wishes to use. Storing absolute path names in client programs is inappropriate as file path names can change. Furthermore, in a heterogeneous or non-uniform environment, absolute path names for servers must differ. Therefore, indirect server referencing is a requirement of remote procedure call systems.

The PARPC system has two conflicting goals with regards to the placement of server files: indirect server referencing and user control. One way to implement indirect server referencing is to force the servers to be placed in a specific directory on each machine. Such an approach is used by Courier.[Coop83] However, a common shared directory must be maintained by a system administrator. Furthermore, once placed in a specific directory, the user has no control over his program. We have found an interesting solution which supports both indirect server referencing and direct user control of server files.

Our first idea was to use hard links to maintain two links to the server. One from the user's directory, and one from a special PARPC server directory. Unfortunately, hard links cannot span file systems. Our solution was to create a special directory on each file system which would contain hard links for user servers within that file system. We then have soft links from a global servers directory into each of the file system specific directories. A PARPC utility called *mkserver* creates the necessary links.

This left one problem unsolved: how to clean up the global and local server directories when the user removes a server file. We chose a straightforward solution. A demon periodically scans the PARPC directories removing any servers which the user has unlinked (i.e. files with one remaining hard link).

With our solution for placing server files, PARPC extends the UNIX development and administration models for distributed programs. Users who develop PARPC programs administer their own files without the support of a system administrator.

4.3. PARPC Administration Tools

The only requirement placed on an administrator is the maintenance of the user equivalence file around the network. However, even this minimal requirement proved to be tedious in the heterogeneous, non-uniform environment at UCSD. To help maintain the equivalence file, we developed two administration tools. The tools are themselves simple PARPC programs. *Reread* executes a parallel procedure at a set of hosts to request that the parent PARPC server reprocess the user equivalence file. This allows network users to be added or removed without having to reboot the servers. *Rgetuid* executes a parallel procedure at a set of hosts to get the local representative of a user and outputs the information in a format suitable for the user equivalence file. Just as the program development tools support distributed program development from a single location in a heterogeneous, non-uniform environment, *reread* and *rgetuid* support administration from a single location.

By extending the UNIX model of access control, resource accounting, program development and administration, our experience has been that PARPC administers itself.

5. Performance Measurements of the PARPC System

The PARPC runtime environment has good performance, especially in light of the fact that it was not implemented in the kernel and that it provides a high level model of computation.

Figure 6 gives timings for null procedure invocation and return using PARPC. PARPC transparently supports three types of procedure calls. *Remote procedure call* is a procedure call made from a process on a local machine to another process on a remote machine. A *local remote procedure call* is made between two processes on the same machine. A *local PARPC procedure call* preserves PARPC semantics and syntax but is made within the same local process. Timings for initial calls are given for both the resident and user server models. Timings for subsequent calls are the same for both server models. Timings for C function calls are also included for reference.

Null procedure invocation and return	Milliseconds	95% C.I.
Subsequent local remote procedure call	5.766	±.008
Subsequent remote procedure calls	6.285	±.008
Initial remote procedure call to resident server	444.22	±12.8
Initial local remote procedure call to resident server	435.47	±16.4
Initial remote procedure call to user server	634.06	±19.4
Initial local remote procedure call to user server	677.26	±20.6
Local PARPC procedure call	0.139	±.001
C function call	0.00537	±.00001

Figure 6: PARPC Communication Overhead Between Two Sun 3/60 Machines (Average of 128 trials)

The timings reflect communication overhead between two SUN 3/60 workstations connected by an ethernet.

Subsequent procedure invocations execute in approximately 6 milliseconds. For comparison, Sprite OS designers report that an inter-kernel null procedure invocation and return between two SUN 3/75 workstations takes 2.8 milliseconds. [Oust88] The timings for Sprite only include execution in the kernel. Our figures reflect a round trip between two user processes.

Invoking the first procedure is substantially more expensive than invoking subsequent procedures because of overhead executing TCP/IP connection protocols, host name resolution, PARPC authentication protocols and remote process initiation. The initial call executed at a resident server does not require invoking a remote process; the first procedure is invoked at a resident server in approximately 70% of the time it takes to invoke the first procedure at a user server. The startup time is not a problem for interactive programs and long lived programs. For programs that make a lot of parallel procedure calls, the initial cost is amortized over future calls.

6. Conclusions

We have described our experiences in designing and using the PARPC system. A program making parallel procedure calls executes as a *network process* and has good communication performance.

PARPC supports heterogeneous, non-uniform environments by extending the simple, well understood UNIX model for local program development and execution and system administration. Because the system uses existing UNIX mechanisms to extend the model, our experience has been that PARPC is easy to use and administer.

References

[Bern83] Bernstein, P. and Goodman, N. ‘‘The Failure and Recovery Problem for Replicated Databases.’’ *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing,* Montreal, 1983. pp 114-122.

[BiNe84] Birrell, A. and Nelson, B. ‘‘Implementing Remote Procedure Calls.’’ *ACM Transactions on Computer Systems,* Vol. 2, No. 1 (February 1984) pp 39-59.

[BuMaPa87] Burkhard, W. A., Martin, B. E. and Paris, J. F. ‘‘The Gemini Replicated File System Test-bed.’’ *Proceedings of the Third International Conference on Data Engineering,* Los Angeles, California.

[CaLoPa87] Carroll, J. L., Long, D.D.E, and Paris, J.F., ‘‘Block Level Consistency off Replicated Files.’’ *Proceedings of the Seventh International Conference on Distributed Computing Systems,* West Berlin, West Germany.

[ChZw85] Cheriton, D. R. and Zwanepoel, w. ‘‘Distributed Process Groups in the V Kernel.’’ *ACM Transactions on Computer Systems,* Vol. 3, No. 2 (May 1985) pp. 77-107.

[Coop83] Cooper, E. ‘‘Writing Distributed Programs with Courier.’’ UNIX Programmer’s Manual, 4.2 Berkeley Software Distribution, Computer Systems Research Group, Computer Science Division, University of California, Berkeley, August, 1983.

[Coop85] Cooper, E. ‘‘Replicated Distributed Programs.’’ Ph.D. Thesis. Technical Report UCB/CSD 85/231. University of California, Berkeley, May, 1985.

[Feld79] Feldman, S.I. ‘‘Make -- A Program for Maintaining Computer Programs."’’ UNIX Programmer’s Manual, Jan. 1979, Bell Laboratories.

[Giff79] Gifford, D. K. ‘‘Weighted Voting for Replicated Data.’’ *Proceedings of the Seventh ACM Symposium on Operating System Principles,* 1979, 150-161.

[John79] Johnson, S.C., ‘‘Lint, a C Program Checker.’’ UNIX Programmer’s Manual, Bell Laboratories.

[KeRi78] Kernighan, B. W. and Ritchie, D. M. ‘‘The C Programming Language.’’ Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[Martin86] Martin, B. E. ‘‘Parallel Remote Procedure Call: Language Reference and Users’ Guide.’’ Technical Report CS-097, UCSD Department of Electrical Engineering and Computer Science, July, 1986.

[Martin87] Martin, B. E. ‘‘Modeling Concurrent Activities with Nested Objects.’’ *Proceedings of the Seventh International Conference on Distributed Computing Systems,* West Berlin, West Germany.

[Martin88] Martin, B. E., ‘‘Leaf Scheduling of Shared Nested Objects.’’ Technical Report CS-094, UCSD Department of Computer Science and Engineering. January, 1988.

[MaBeRu87] Martin, B. E., Bergan, C.A., and Russ, Brian, ''PARPC: A System for Parallel Procedure Calls'' *Proceedings of the 1987 International Conference on Parallel Processing,* The Pennsylvania State University Press.

[MeBo76] Metcalfe, R. and Boggs D. ''Ethernet: Distributed packet switching for local computer networks.'' *Communications of the ACM*, July 1976.

[Oust88] Ousterhout, J., Cherenson, A., Douglis, F., Nelson, M., Welch, B., ''The Sprite Network Operating System'' *Computer*, February 1988.

[Saty86] Satyanarayanan, M. *RPC2 User Manual.* Rep. CNMU-ITC-84-036, Information Technology Center Carnegie-Mellon U. (July 1986).

[Stro86] Stroustrup, B. ''The C++ Programming Language.'' Addison-Wesley, 1986.

[Sun84a] Sun Microsystems, ''External Data Representation Reference Manual.'' Mountain View, California, October, 1984.

[Sun84b] Sun Microsystems, ''Remote Procedure Call Reference Manual.'' Mountain View, California, October, 1984.

[TaRe85] Tanenbaum, A. S. and R. van Renesse, ''Distributed Operating Systems,'' *ACM Computing Surveys* 17, 4 (Dec. 1985), 419-470.