CONF-9010256--2

DISCLAIMER

WSRC-MS--90-217

DE91 005120

CLIENT/SERVER DISTRIBUTED PROCESSING USING SUNOS REMOTE PROCEDURE PROTOCOL (U)

by

K. E. Hammer and T. L. Gilman

Westinghouse Savannah River Company
Savannah River Site
Aiken, South Carolina 29808

A paper proposed for presentation and publication at the
Westinghouse Computer Symposium
Pittsburgh, PA
October/November 1990

This paper was prepared in connection with work done under Contract No. DE-AC09-89SR18035 with the U.S. Department of Energy.

CLIENT/SERVER DISTRIBUTED PROCESSING USING SUNOS REMOTE PROCEDURE PROTOCOL (U)

by

K. E. HAMMER AND T.L. GILMAN
WESTINGHOUSE SAVANNAH RIVER COMPANY
SAVANNAH RIVER LABORATORY
AIKEN, SC 29808
(803) 725-1611

ABSTRACT

Over the last ten years the development of PC's and workstations has changed the way computing is performed. Previously, extensive computations were performed on large high speed mainframe machines with substantial storage capacity. Large capital and operational costs were associated with these machines. The advent of more powerful workstations has brought more computational cycles to the users at lower cost than was achieved with busy timesharing systems. However, many users still can't afford individual special purpose hardware or gigabytes of storage. A successful distributed processing environment must share these resources.

Client/Server models have been proposed to address the issues of shared resources. They are not a new idea, but their implementation has been difficult. With the introduction of SUN's public domain Remote Procedure Call (RPC) Protocol and SUN's interface generator, RPCGEN, their implementation has been made easier. SUN has developed a set of "C" callable routines that handle the Client/Server operations. The availability of Network File System (NFS) on the SRL CRAY and the arrival of Wollongong's latest version of NFS has allowed applications to be ported easily. Only recompiling is required to allow resource and information sharing between computing platforms.

This paper reviews the Client/Server model with respect to SUN's RPC implementation. The discussion will focus on the RPC connection between local and remote machines, the RPC Paradigm for making remote procedure calls, and the programming levels of the RPC libraries. The paper will conclude with summaries of two applications developed at SRL using the protocol and their effect on our computing environment. These include the Nuclear Plant Analyzer and an animation of fluids using behavioral simulation of atom-like particles.

0. Introduction

Over the last ten years the development of PC's and workstations has changed the way computing is performed. Previously, extensive computations were performed on large high speed mainframe machines with substantial storage capacity. Large capital and operational costs were associated with these machines . The advent of more powerful workstations has brought more computational cycles to the users at lower cost than was achieved with busy timesharing systems. However, many users still can't afford individual special purpose hardware or gigabytes of storage. A successful distributed processing environment must share these resources.

Client/Server models have been proposed to address the issues of shared resources. They are not a new idea, but their implementation has been difficult. With the introduction of SUN's public domain Remote Procedure Call (RPC) Protocol and SUN's interface generator, RPCGEN, their implementation has been made easier. SUN has developed a set of "C" callable routines that handle the Client/Server operations. The availability of NFS on the SRL CRAY and the arrival of Wollongong's latest version of Network File System (NFS) has allowed applications to be ported easily. Only recompiling is required to allow resource and information sharing between computing platforms.

The RPC developed by Sun Microsystems relieves programmers from concerning themselves with the detailed aspects of UNIX sockets programming. In addition, SunOS includes a set of External Data Representations (XDR) for the transmission of data across sockets. The combination of RPC and XDR allows programmers to implement Client/Server connections between disconnected processes with minimal effort. The programming of RPC/XDR applications is thoroughly documented in the SunOS Network Programming Manual [1]. The following discussion will not cover detailed programming aspects of RPC/XDR programming, but will present the algorithms by which RPC connections are made and XDR transmissions are accomplished.

1. RPC Interprocess Connections

The object of RPC connections is to allow local user applications to access remote services, possibly user defined, by establishing a Client/Server relationship with a remote host. The remote host can be any machine where RPC/XDR libraries are available. In the interest of debugging the application, Client and Server applications could reside on the local machine. The following discussion traces SUN's procedure for establishing the Client/Server connection.

2. Installation of Remote Services

Before connections are possible, the remote service programs must have been installed with the portmapper on the remote machine. Each service program is installed on its own port and contains multiple versions of each of the service

procedures. The service procedures are "C" callable functions that use the address of a single argument, either a value or structure, and return the address of a single result. either a value or structure. Sun has simplified this process by suppling an interface generator.

The Client/Server interface routines for these functions may be generated using RPCGEN and thus may be modified by the programmer if necessary. RPCGEN is a SunOS application that uses "C" like XDR constructs to define argument and result interfaces for each of the service procedures (RPC.x). RPCGEN generates four files from the interface definitions: the interface "C" structures (RPC.h), the interface encode/decode filters (RPC_xdr.c), the Server installation program (RPC_svc.c), and the service procedure stubs for the Client application (RPC_clnt.c). The Server installation program, XDR filters, interface structures, and service procedure functions are moved to the remote machine where they are compiled and linked with the RPC/XDR libraries. These steps are shown in Figure 1. Note that in the figure the CRAY is the Client and the SUN is the sever.

The installation program (RPC_svc.c) preforms the following functions. It ensures that the old copies of the programs and versions to be installed have been removed from the portmapper registers each of the service programs and all their versions, and executes a Server loop that waits until the portmapper receives a request for its services. Installation of service programs is denoted by step 0 in Figure 2. Note that each service program can have multiple version which allows modifications to be made without affecting access to older versions of the function. Making connections and using these services is now possible.

3. The Remote Procedure Call Process

The RPC process is a four-step procedure that has been diagrammed in Figure 2. Steps 1 throgh 4 include contacting the portmapper, return the port of the requested service program and version, calling the remote service procedure on the returned port, and then waiting to receive the results. Each of these steps will be considered in more detail.

Making the connection between the Client and the Server, step 1 and 2, is performed by procedures supplied with the RPC/XDR implementation. They are not limited to making connections to the portmapper, but can also connect to files, pipes, or memory. For Client/Server applications, the local procedure contacts the remote portmapper with the program and version number (constants define in RPC.h) of the services to be requested. The portmapper returns the port of the requested services to the procedure. The procedure builds a Client structure that contains the remote port address, the instructions for the access the specific connection and the current operating conditions of the Client/Server interface.

The programmer uses a remote service by calling the remote service stub provided in RPC_clnt.c. Four changes have been made to the calling structure

of all service procedures. The calling structures of the services have been modified by RPCGEN. The version number is first appended to the the procedure name (see Figure 1). The Client structure has been added to the argument list and the address of the argument structure is passed in place of the structure itself. This passes the Client and argument structure to the port address returned earlier. Note that the Client application now waits until the remote service has completed and returned control to the Client. This is important when considering RPC for distributed parallel applications. When the remote service procedure has completed, the result structure is passed back across the port connect and returns control to the Client application. This process is represented by step 3 and 4 of Figure 2 and will be discuss in further detail in the next sections.

Using RPCGEN is advantageous because the Client/Server model just presented can be implemented by writing the RPC.x file and its remote service procedures. This is an excellent method for interface control as well.

4. The RPC Paradigm

The RPC paradigm defines the protocol used to perform remote service procedure execution using SunOS Remote Procedure Calls. This process has been diagramed in Figure 3. The operation performed is divided into five steps including local CALLRPC function, invocation of remote services, service procedure execution, request completion, and return reply. Each of these steps will be discussed.

CALLRPC Function

The CALLRPC function appears as a function call within the Client application. Its arguments includes the address of the argument structure and the RPC Client structure. This function call is one of the Client stubs produced by RPCGEN. The stub performs two functions. It sets the static return structure to zero and performs a Client call to the Server. The Client call passes as its arguments the Client structure, the remote service procedure number for the function called, the XDR translation filter for the argument structure, the address of the argument structure, the XDR translation filter for the return structure, the address of the return structure, and a time-out structure. The Client call encodes the Client and argument structures onto the port connection. The argument translation is performed with the XDR filter generated by RPCGEN. The function waits for control to be returned from the Server. The Server will return control when it has finished processing the service procedure. Note that if no reply is received within the time-out period, an error is returned by the function call.

Invocation of Remote Service

The Server has been asleep since it was installed. It is awakened by data appearing at its port. Svc_run performs this function. It was called during

installation, but should never return. The svc_run routine decodes the Client structure from its port and calls the installed Server routine (RPC_svc.c).

Service Execution

The RPC_svc.c remote Server routine was generated by RPC. It is a list of case statements, one for each service procedure associated with this program and version number. Each case assigns the argument filter, the service procedure, and the return filter to default variables. The case statements are concluded and the defaults are executed. First, the argument structure is decoded from the port, then the remote service procedure is called, and after execution has completed the return structure is encoded onto port. It should be noted that decoding of the argument structure may have caused memory to be allocated on the remote machine. It not freed, memory allocations will accumulate each time the service is called. Thus, the service execution should always free memory allocated for the argument structure. When these operations have completed, control is returned to the service deamon.

Request Completion

The request completion function of the service deamon does little at this point. It puts the Server back into a wait state and returns control to the Client.

Return Reply

The return reply performs three functions. It awakens the Client for processing when port action is detected. Next, it decodes the return structure from the port using the XDR filter and passes the the address of the return structure to the Client.

5. Three Levels of RPC

Sun divides RPC into three levels. The first level is not really a part of RPC, but is rather how to use RPC. This is the sharing of information and computing cycles between heterogeneous machines. This level keeps the user isolated from network programming by presenting RPC services as callable routines in link libraries. The second level of RPC is the process shown in Figure 1 which has already been functionally described. This level provides the programmers with simple access to RPC communication, allowing them to install user services with the remote portmappers and to develop local Client applications for using these remote services. The third level is the nuts and bolts of programming the RPC interface. This level gives the programmer greater control over the RPC/XDR connection and allows for development of new applications based on the tools provided in the RPC libraries. The programmer has control over the following areas:

* TCP vs. UDP Network Protocols
* Memory Allocation during XDR
* Authentication

* Time-out Control
* Selection of Sockets
* Broadcast RPC/Batching
* Callback Processing
* Port, File, Pipe, and Memory Input/Output
* Control of Direction (Encode, Decode, and Free)
* Advance Data Structures

The last area is accomplished with a rich set of XDR primitives provided in the RPC libraries and documentation for programmer use. These primitives include word and char, floating point, enumeration, void, and constructed data type filters. The constructed data types are build on the earlier primitives and are included in the library for user convenience. It is important to note that the XDR format produced by encoding the data is the same on other systems. Thus any system can decode XDR information encoded by any system. This simplifies the sharing of information between different machines because the software used to generate the information is directly portable to any machine that needs to use the information. This is why RPC/XDR was selected for our projects.

6. RPC/XDR Projects at Savannah River Laboratories

Two projects that use the SunOS Remote Procedure Calls and External Data representations will be discussed in the following sections. TRANS_RPC is a distributed process that translates and transfers TRAC graphics information from the UNICOS CRAY to the Sun Workstations. The other project is a computational Server on the UNICOS CRAY that does particle motion simulation for a Sun Workstation Client who is feeding the particle positions to a PIXAR for rendering. The PIXAR performs the rendering independent of the workstation and allows some parallel processing to occur. Each project will be presented in more detail.

7. TRANS_RPC Project

TRANS_RPC is a sub-task of the Nuclear Plant Analyzer (NPA) currently under development at Savannah River Laboratory. The NPA is a graphical user interface that allows the user to analyze data being generated remotely. Currently, the remote source of information is the thermal hydraulics code TRAC running on our UNICOS CRAY. TRANS_RPC functions to transport the remote information to the local user. The following goals were set for this task.

* Capture of TRAC Graphics Data
* Convert TRAC/CRAY Information in PTR/GRF/SUN Format
* Convey PTR/GRF/SUN formatted Data to SUN Workstation
* Concurrent Performance of Tasks

The original method for meeting these goals is presented in Figure 4. The process required four user steps. The user began by generating the TRAC results in the form of a TRCGRF binary file. This file was than processed by SUNGRF which decoded the binary information into an ASCII pointer file

(TRCGRF.PTR) and the time dependent binary graphics file (TRCGRF.GRF). It should be noted that the TRCGRF.GRF is formatted IEEE 32-bit float point values with UNICOS blocking words inserted at regular intervals. Transferring these file was accomplished using FTP between the CRAY and the Sun Workstation. The final step was unblocking the TRCGRF.GRF file by removing the UNICOS blocking words. This method was abandoned because it was cumbersome to the user and did not lend itself to concurrent performance.

TRANS_RPC was developed to replace the previous method. The first step in in the process was designing the RPC interfaces using RPCGEN (Figure 1). The next step required being able to read UNICOS binary files and named pipes with a "C" program. "C" had to be used because it would properly interface with a FIFO named pipe which automatically synchronizes data transmission between concurrently running processes. The development of the routines for processing the binary information into the PTR and GRF argument structure, and the routines for writing these structures in TRCGRF.PTR and TFRGRF.GRF file formats, were performed in parallel (Figure 5). This provided a quality assurance check at each step in the development. When the CRAY program could process the entire TRCGRF file correctly, the structure writing routines were replaced with their equivalent RPC stubs and the routines were modified to be used as remote procedure services (Figure 6). These modified routines were moved to the Sun Workstation and linked into the remote Server. The Client/Server application was then tested using both named pipes and previously produced TRCGRF files of various sizes. All tests have been successful.

The results of this project have been good. It has provided faster access to the information, a simpler user interface between the CRAY and Sun Workstation, and automatically reformatted information from 64-Bit CRAY words to 32-Bit SunOS words. In addition, it has shown how to build Client/Server shells around large programs on remote machines and transport the necessary I/O to local workstations, giving the illusion to the user that the program is being run locally.

8. Simulation of Liquid Across a Sun/Cray RPC Link

A method was developed to create animations of liquids using behavioral simulations [2]. The behavioral simulation relied on representative "super-atoms" to simulate the motion of liquids. These super-atoms had similar nuclear properties to normal atoms, however, one super-atom represented millions of normal atoms. The original animation code was developed on a Sun 3/280 which was acting as a host for a Pixar Image Computer. The Sun was used to determine the motion of individual atoms of the liquid while the Pixar was used to render the surface described by the atoms.

Due to the number of atoms which were used in the simulation, the Sun quickly proved ineffective as a motion control engine. However, since the Sun was the host for the Pixar it was necessary for the Sun to have some part of the image generation process. The use of RPC calls to a Cray X/MP was picked as a

method which would allow the complex molecular interactions to be calculated quickly while still allowing the Sun/Pixar to render the liquid surface.

The code was then designed so that it ran on three separate machines: a Cray X/MP, a Sun 3/280, an Abekas A60, and a Pixar Image Computer. This was done to optimize the code fully so that reasonable turnaround times for animations could be achieved. The Cray and Sun are linked using RPC with XDR protocol to pass data between them. The Sun and Pixar are linked through Pixar's proprietary software using a high speed data link.

The driver program resides on the Sun. This program is responsible for initializing all of the super-atoms, as well as setting up links to the Cray, Pixar, and Abekas A60. The user interface resides on the Sun as well. RPC calls are made to the Cray to do the super-atom motion calculations. After each time step calculation, the Pixar is called to render the new set of positions of the liquid super-atoms. Once the Pixar renders the image, the Sun directs the A60 to record the frame. This provides a smooth, heterogeneous computing environment for producing animation of the simulated liquids.

Figure 7 shows the flow of program control and data for the animation system. As shown, most of the Sun's work involves initialization and control. The Cray is called before and during the animation loop. Whenever possible the code runs in parallel. The call to the ChapReyes subroutine (Chreyes in the diagram) causes the Pixar to begin rendering the image. Control, however, is immediately returned to the Sun when the rendering process begins. At this point the Sun is free to do anything which does not affect the rendering process. In particular, the Sun calls the Cray to generate the super-atom position for the next time step. These operations occur in parallel. Once the Cray RPC returns, the Sun waits for the Pixar to finish so that it can tell the A60 to record the frame.

There are many advantages to this system. First, by breaking the different routines down into ones which can be spread out over a network of machines, it is possible to get better overall speed performance. This is shown effectively in the current implementation. The motion routines could easily be moved to more massively parallel machines which could capitalize on the parallel nature of the code. Distributed computing is one of the methods used to increase the total productivity of a system [3]. Given the existing code, it would be fairly easy to swap routines onto faster or more suitable computers.

The next advantage of this type of approach is that the front end (user interface) can be placed on a machine which is well suited for this job. For instance, it would not be as logical to put the user interface on the Cray, since the Cray is better suited for number crunching. The user interface, shown here, does not pull CPU power away from the number crunching machines (Cray and Pixar).

An advantage which deals primarily with the modular nature of the code is the idea of routine substitution. Different types of liquid dynamics codes can quickly be interfaced into this system as long as the new code conforms to the data protocols set up by this paper. Without using RPC it is possible to have these

alternative dynamics codes run separately and dump their output to a file. Depending on the number of particles in the system, however, this could prove to be very costly in terms of disk storage. Using the RPC approach eliminates the need for massive disk storage since each step of the animation requires only the current data coordinates. There is no need to store these values on disk.

9. Conclusions

An algorithmic discussion of Client/Server model implemented with SunOS Remote Procedure Call has been presented. The discussion reviewed the RPC process, the SunOS RPC paradigm, and the various levels of RPC programming. Two projects using RPC were summarized. They both demonstrated many of the advantages of using the RPC/XDR protocols. The primary advantage is the application of a Client/Server model between remote machines that are distinctly different. The primary disadvantage is the transfer of control while remote processing occurs. This hinders the direct application of RPC to parallel processing. However, as both applications demonstrated, there are ways to work around this limitation. In the future, RPC protocols will continue to be an integral part of the NPA and will be used as a centralized Server of huge databases that cannot be stored locally.

References

[1] *Networking Programming*, Sun Microsystems, Rev A, May 1988.

[2] T. L. Gilman, *Animation of Fluids Using Molecular Behavioral Simulation.* Masters Thesis, University of South Carolina (1990).

[3] Friedman, D., Wise, D., *Aspects of Applicative Programming for Parallel Processing*, IEEE Transactions on Computers, Vol c-27, No. 4, April 1978.

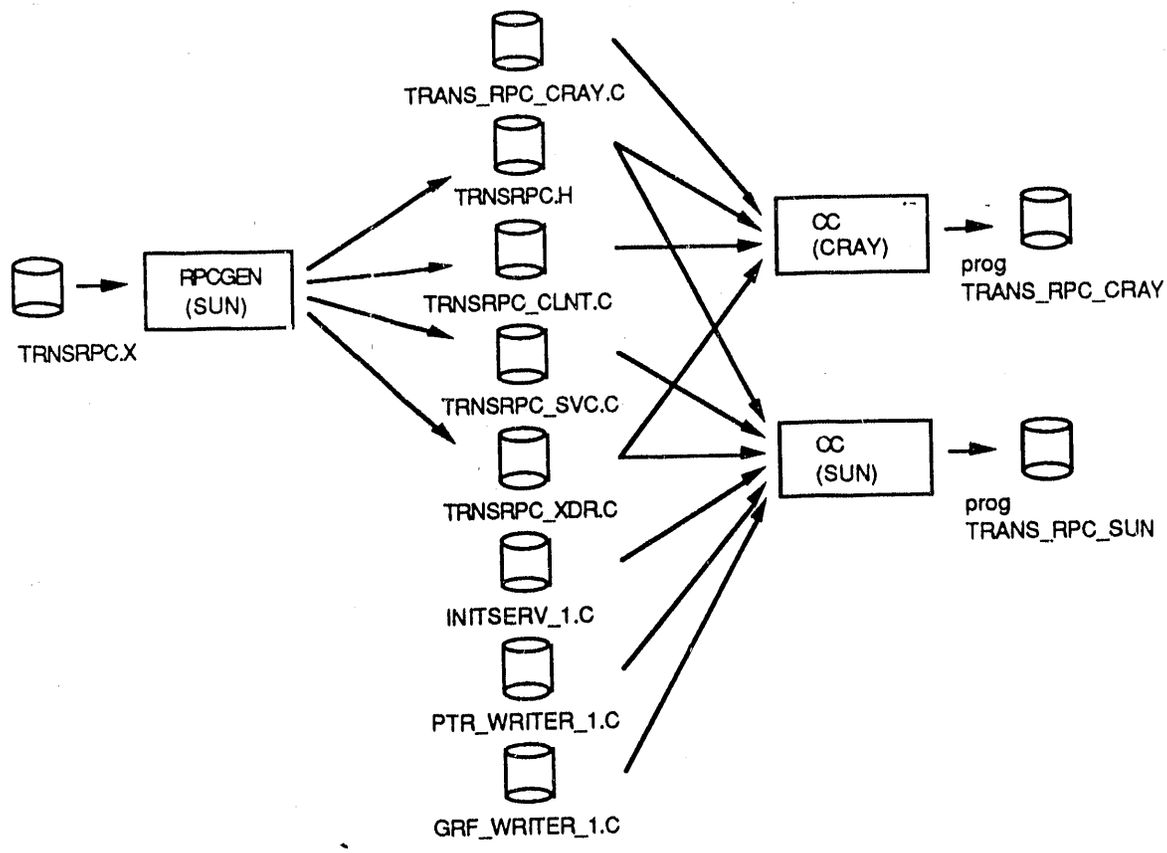

Figure 1 Compiling an RPC Application

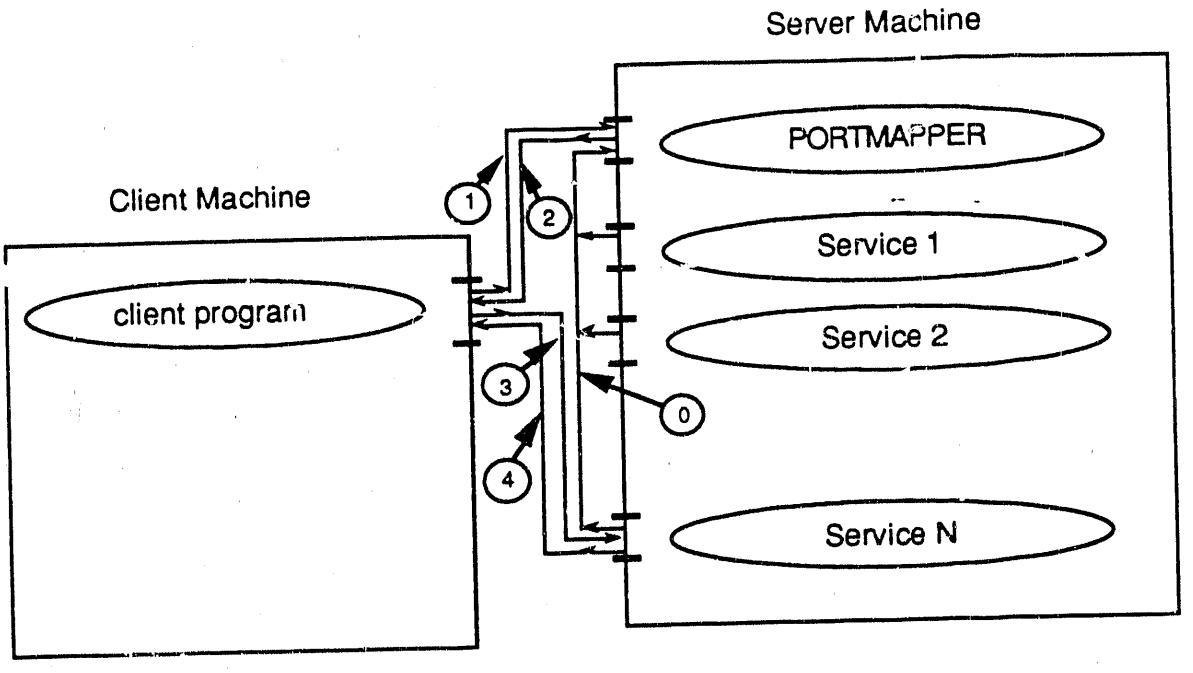

Figure 2 The RPC Process

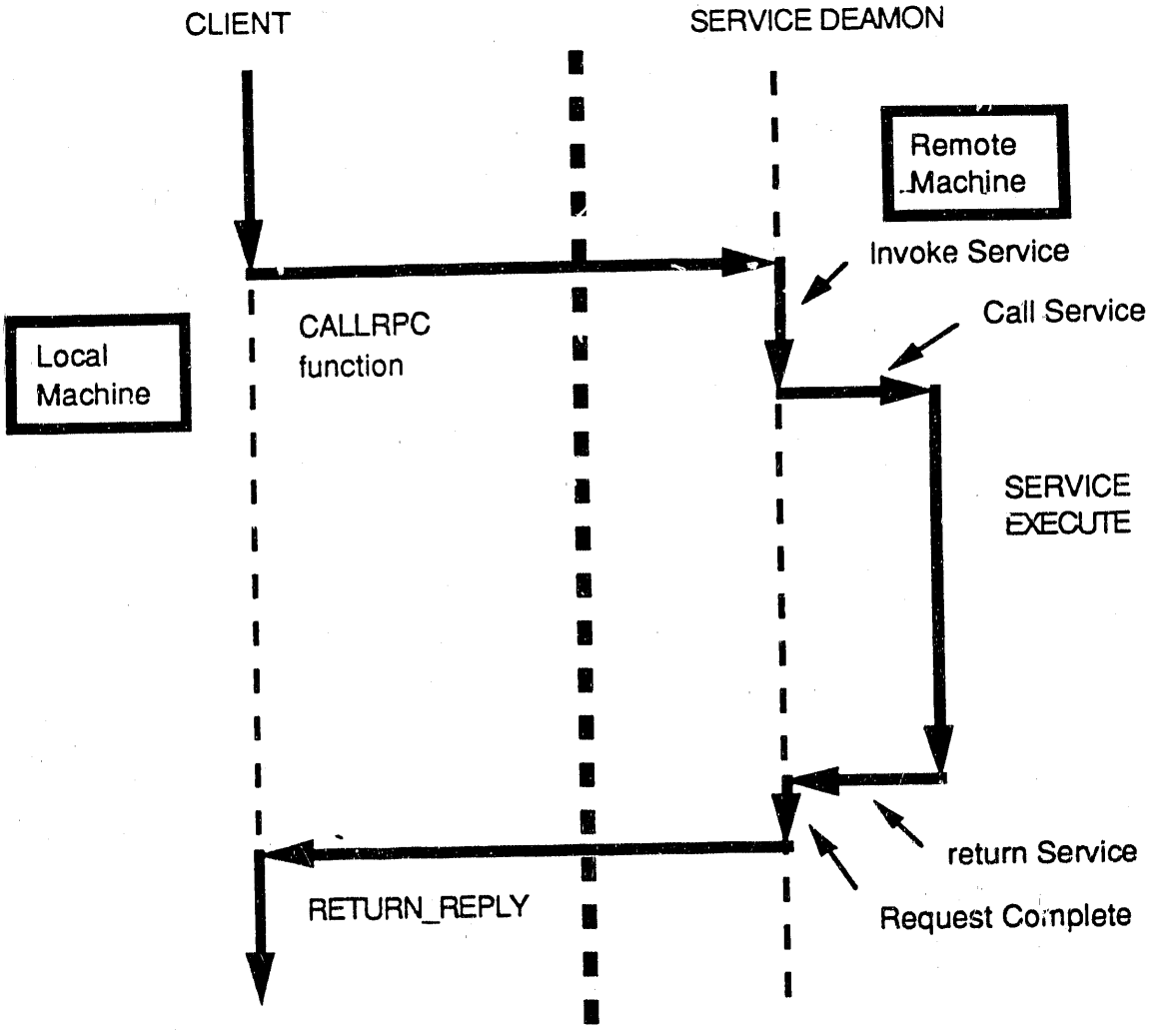

Figure 3 The RPC Paradigm

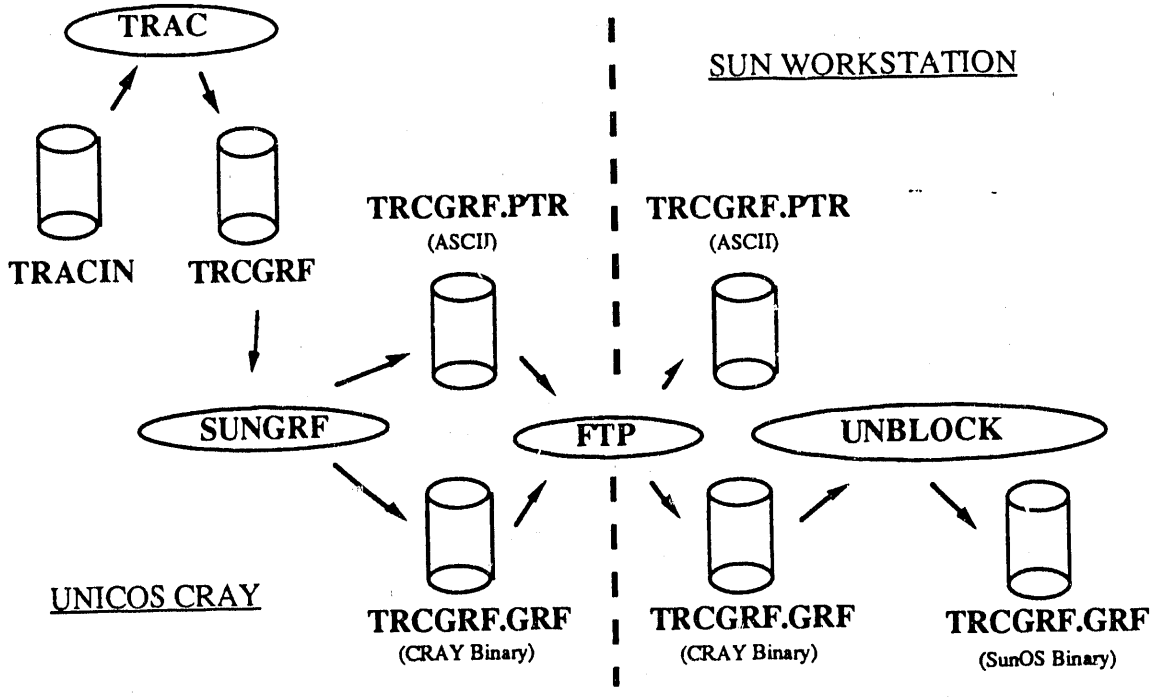

Figure 4 The Manual Method

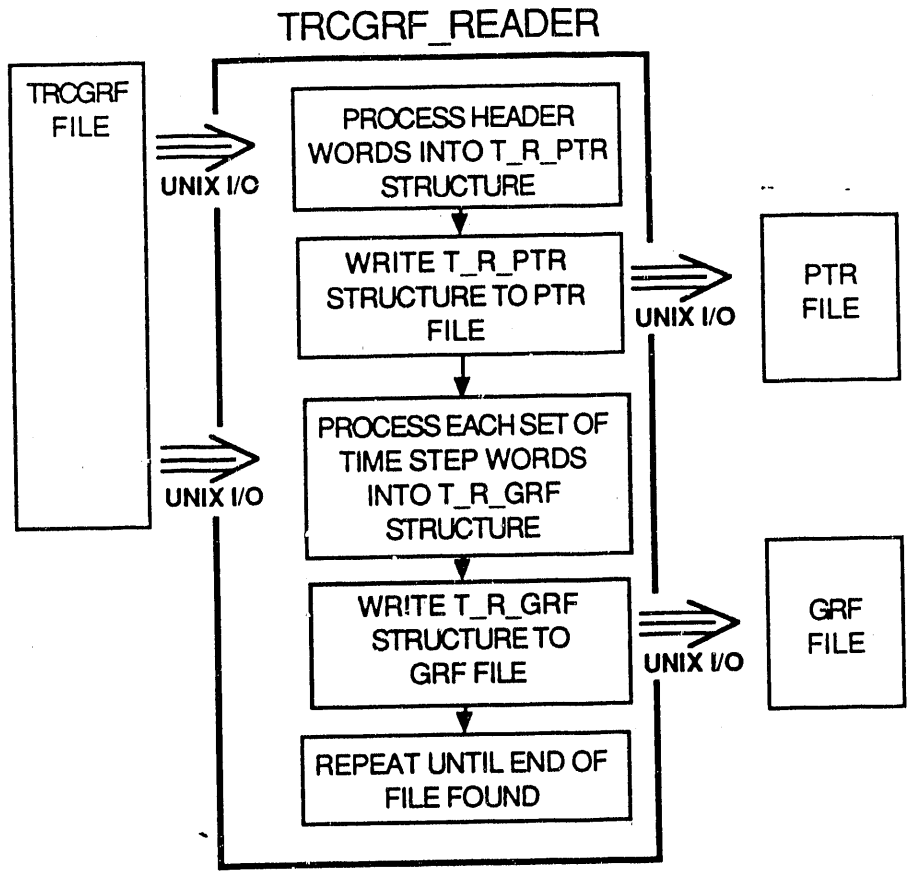

Figure 5 Development and Testing Program

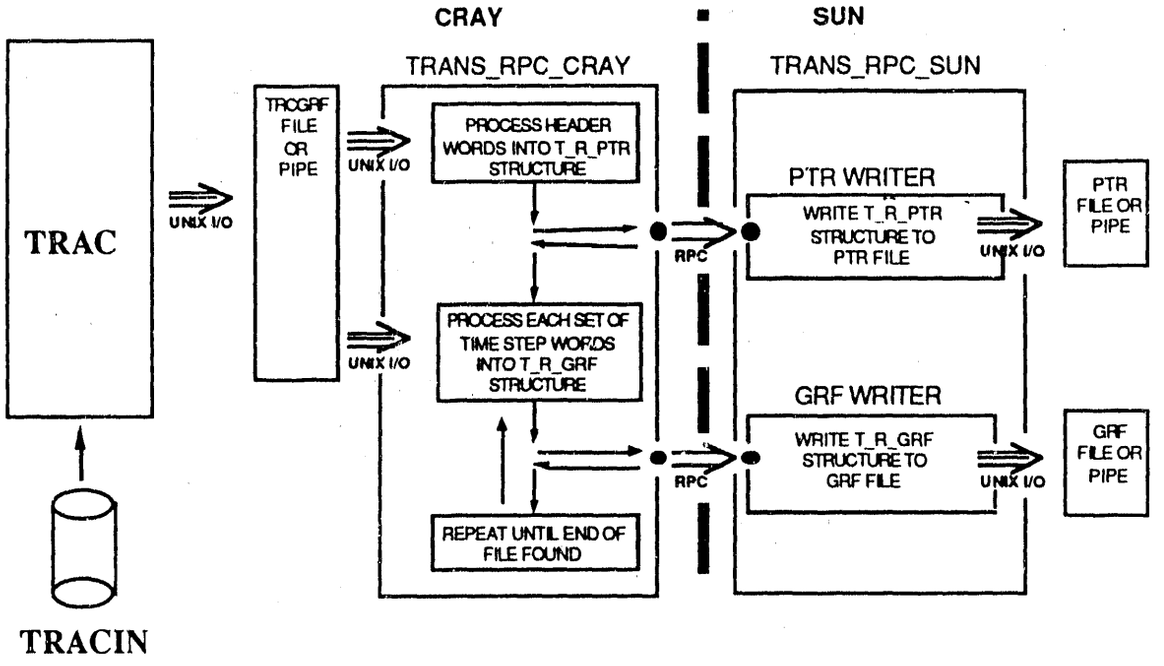

Figure 6 TRANS_RPC Information/Control Diagram

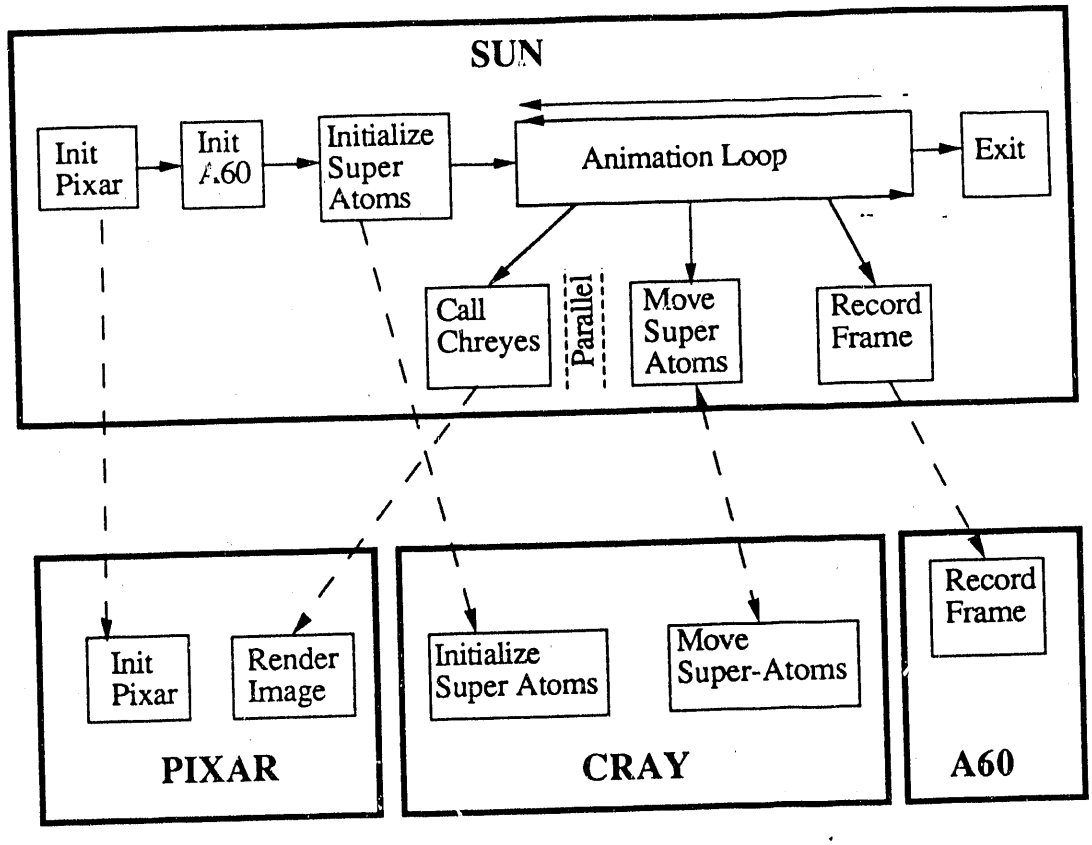

Figure 7 Particle Motion Simulator Program-Data Flow Graph

END

DATE FILMED

12/31/90