REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (*Leave blank*)

2. REPORT DATE
Dec 94

3. REPORT TYPE AND DATES COVERED

4. TITLE AND SUBTITLE
A Demonstration of Client/server Technology Using Remote Procedure calls with an application of File migration for Moving Records based on Time

5. FUNDING NUMBERS

6. AUTHOR(S)
Andrew C. Jank

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
AFIT Students Attending:
Arizona State University

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/CI/CIA
94-155

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
DEPRTMENT OF THE AIR FORCE
AFIT/CI
2950 P STREET
WRIGHT-PATTERSON AFB OH 45433-7765

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for Public Release IAW 190-1
Distribution Unlimited
MICHAEL M. BRICKER, SMSgt, USAF
Chief Administration

12b. DISTRIBUTION CODE

13. ABSTRACT (*Maximum 200 words*)

DTIC
ELECTE
JAN 0 5 1995
S F

19950103 044

DTIC QUALITY INSPECTED 3

14. SUBJECT TERMS

15. NUMBER OF PAGES
60

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT

18. SECURITY CLASSIFICATION OF THIS PAGE

19. SECURITY CLASSIFICATION OF ABSTRACT

20. LIMITATION OF ABSTRACT

ABSTRACT

As the computing resources of the world are quickly becoming integrated into a vast network of interconnected machines, client/server technology is becoming more prevalent as a method to utilize unused resources and to distribute complex systems. Remote Procedure Calls (RPCs) serve as one of the means to implement this technology. This report, combined with those produced by 2Lt Andrea Miller and 2Lt Eric DeLange, provide an overview of both the complexity and the benefits of using RPC in a distributed environment. The report initially investigates the fundamentals of RPC and follows with a discourse of RPC's suitability for implementation as a means for computer-integrated file migration. The report presents an application involving RPC for timed file migration and concludes with a discussion of implementing user-friendly front-ends to hide the complex nature of RPC from the user. This specific application investigates the applicability of RPC to developing a distributed file migration application. The application allows the user to access files that may reside on various hosts by querying a central database for a file's location via RPC. Some of the files, however, are dynamically relocated, based on a timing procedure. This could be very advantageous for global organizations that maintain a core set of organization files that must be accessed at a specific time of day (which, of course varies, depending on the time zone of the requester).

Title:	A Demonstration of Client-Server Technology Using Remote Procedure Calls With an Application of File Migration for Moving Records Based on Time
Author:	2Lt Andrew C. Jank, USAF
Date:	December 1994
Pages:	60
Degree:	Master of Science in Decision and Information Systems
Institution:	Arizona State University

ABSTRACT

As the computing resources of the world are quickly becoming integrated into a vast network of interconnected machines, client/server technology is becoming more prevalent as a method to utilize unused resources and to distribute complex systems. Remote Procedure Calls (RPCs) serve as one of the means to implement this technology. This report, combined with those produced by 2Lt Andrea Miller and 2Lt Eric DeLange, provide an overview of both the complexity and the benefits of using RPC in a distributed environment. The report initially investigates the fundamentals of RPC and follows with a discourse of RPC's suitability for implementation as a means for computer-integrated file migration. The report presents an application involving RPC for timed file migration and concludes with a discussion of implementing user-friendly front-ends to hide the complex nature of RPC from the user. This specific application investigates the applicability of RPC to developing a distributed file migration application. The application allows the user to access files that may reside on various hosts by querying a central database for a file's location via RPC. Some of the files, however, are dynamically relocated, based on a timing procedure. This could be very advantageous for global organizations that maintain a core set of organization files that must be accessed at a specific time of day (which, of course varies, depending on the time zone of the requester).

BIBLIOGRAPHY

Bloomer, John. Power Programming with RPC. Sebastopol: O'Reilly and Associates, Inc., 1991.

Carpenter, B. E. and R. Cailliau. "Experience with Remote Procedure Calls in a Real-time Control System." Software--Practice and Experience. Vol. 14. September 1984, p. 901-07.

Comer, Douglas E. and David L. Stevens. Internetworking with TCP/IP. Vol 3. Inglewood Cliffs: Prentice Hall, Inc., 1993.

Curry, David A. Using C on the UNIX System. Sebastopol: O'Reilly and Associates, Inc., 1989.

Hac, Anna. "A Distributed Algorithm for Performance Improvement Through File Replication, File Migration, and Process Migration." IEEE Transactions on Software Engineering. Vol 15, No 2. November 1989, pp. 1459-1470.

Hahn, Harley. A Student's Guide to UNIX. New York: McGraw-Hill, 1993.

Hewlett-Packard Company. HP Visual User Environment 3.0 User's Guide. Hewlett-Packard Company, 1992.

Kernighan, Brian W. and Dennis M. Ritchie. The C Programming Language. 2nd ed. Murray Hill: AT&T Bell Laboratories, 1988.

Korzeniowski, Paul. "Make Way for Data." Byte. June 1993, pp. 113-115.

Levy, Henry M. and Ewan D. Tempero. "Modules, Objects and Distributed Programming: Issues in RPC and Remote Object Invocation." Software--Practice and Experience. Vol. 21. January 1991, pp. 77-90.

Soto, J. L. Cruz, M. C. Calzada Canalejo and M. Marin Beltran. "Parallelization of Differential Problems by Partitioning Method (Synchronized Algorithm)." Computers and Mathematics with Applications. July 1993, pp. 25-31.

Sun MicroSystems. Network Programming Guide. Sun MicroSystems, Inc., 1990.

Waite, Mitchell and Stephen Prata. New C Primer Plus. Carmel: Sams Publishing, 1993.

A Demonstration of Client-Server Technology Using Remote Procedure Calls With an Application of File Migration for Moving Records Based on Time

by

2Lt Andrew C. Jank

An Applied Project Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science in Decision and Information Systems

Arizona State University

December, 1994

EXECUTIVE SUMMARY

As the computing resources of the world are quickly becoming integrated into a vast network of interconnected machines, client/server technology is becoming more prevalent as a method to utilize unused resources and to distribute complex systems. Remote Procedure Calls (RPCs) serve as one of the means to implement this technology. This report, combined with those produced by 2Lt Andrea Miller and 2Lt Eric DeLange, provide an overview of both the complexity and the benefits of using RPC in a distributed environment. The report initially investigates the fundamentals of RPC and follows with a discourse of RPC's suitability for implementation as a means for computer-integrated file migration. The report presents an application involving RPC for timed file migration and concludes with a discussion of implementing user-friendly front-ends to hide the complex nature of RPC from the user. This specific application investigates the applicability of RPC to developing a distributed file migration application. The application allows the user to access files that may reside on various hosts by querying a central database for a file's location via RPC. Some of the files, however, are dynamically relocated, based on a timing procedure. This could be very advantageous for global organizations that maintain a core set of organization files that must be accessed at a specific time of day (which, of course varies, depending on the time zone of the requester).

TABLE OF CONTENTS

PREFACE

This project represents one part of three related and closely coordinated projects. In total, the three projects provide: a) exposition and programming examples needed to understand the use of Remote Procedure Calling (RPC) for distributed Client/Server processing in networks, b) extensions needed for RPC applications that migrate files among hosts in a network based on several interesting and practical criteria, and c) a discussion and code pertaining to the development of a Graphical User Interface (GUI) for the on-line demonstration of concepts spanning the collection of projects.

Working as a team, Eric DeLange, Andrew Jank and Andrea Miller jointly participated in developing a tutorial discussion of RPC concepts and programming techniques. After the development of fundamental RPC programs which illustrate these techniques, the team collaborated to develop a foundational set of code pertinent to file migration among networked hosts. Thereafter, each team member individually expanded this code to model specific RPC applications of file migration.

Finally, a jointly-developed GUI to facilitate the convenient on-line execution and demonstration of all applications was added. Source files and a discussion of the GUI are of considerable instructional interest because they present techniques for synthesizing window programming and C programs that not only make RPC calls but also make system calls to invoke UNIX scripts and utilities.

For the convenience of the reader, each of the related project reports contain the source code and discussion of the applications developed by the companion authors. This project report uniquely contains the discussion and source code developed by the author for an application in file migration based on timed file transfers. Readers who are interested in immediate file migration or migration based on heuristics are free to reference the reports authored by Andrea Miller and Eric DeLange, respectively.

INTRODUCTION

With the vast technological advances in computer capabilities, specifically in the distributed environment, organizations are moving away from centralized systems in order to avail themselves of the various advantages offered by distributed systems. Often, organizations are geographically dispersed which leads them to a natural distribution of computing resources. Furthermore, distribution provides a business with enhanced reliability by allowing data replication across multiple sites. Finally, given the size of organizations in today's business arena, a distributed environment facilitates high transaction rates and also allows for the integration of heterogeneous platforms that often accompany mergers between companies or the addition of new locations.

With the advantages of a distributed environment comes the added complexity of communicating among multiple machines across a network. Since the information is no longer on a single machine, there must be a way to track the identities of the requester and the recipient as well as the location of the information required for the transaction. Client/Server technology is an approach which deals with this complexity. In essence, the requesting computer is viewed as a client which asks for a particular service from another machine (the server). When the client requests a service from the server, control is passed to the server until it has completed processing the request, at which time the client receives the requested information and regains control. For example, if a client needs the result of a complex mathematical function, but lacks the processing power to compute it within a reasonable amount of time, it can request the services of another processor (server) that has the necessary capabilities to perform the calculation. Once the calculation has been performed, the server passes the answer, and control, back to the awaiting client.

The above example looks very much like a function call in any standard language such as C or FORTRAN and, in fact, it is. The only difference is that the function call is made over a network to a different machine. In truth, making a function call to another machine is aptly accomplished through the implementation of Remote Procedure Calls (RPC). RPC is a popular framework for programming in a distributed client/server environment. It provides a means by which a client can communicate with a server.

This project develops an application using RPC. Specifically, it illustrates the usefulness of RPC and its applicability in the area of file migration. In achieving this end, a small, simple program was developed to provide a basic understanding of RPC. Once a working application in the area of file migration was achieved, each member of the group modified the program to accommodate variations in the file migration model. Finally, a front-end was added on the Hewlett-Packard UNIX System to make the program more user-friendly.

REMOTE PROCEDURE CALLS

Remote Procedure Calling permits a client to execute procedures on other networked computers. In fact, RPCs serve as the basis for a majority of the distributed system utilities currently in use (i.e. NFS and NIS). A major reason for RPC utilization is the ease with which RPCs can be implemented, when compared to the lower-level network socket interfaces which have been required prior to the advent of RPCs. Moreover, RPCs are perceived as powerful programming tools, especially for users, since their implementation resembles traditional programming methodologies and the network interfaces are provided with increased transparency to the users (Bloomer 1).

RPC has been identified as a type of middleware. Middleware is software that translates communication between different machines or platforms. This type of software protocol is often necessary for communications within a client/server environment. RPC has been deemed one of the two primary types of client/server middleware; the other is message processing (Korzeniowski 114). In order to use RPC, synchronous links between computers must be established, either using datagram or TCP transports. If no transport is available at the time of initiation, the client application will automatically wait for an answer from the server, and eventually "time out" (halt) when no reply is gathered. Message passing differs from RPC protocol in that messaging systems work on the store and forward principle (Korzeniowski 114). Store and forward systems allow a server to read a computer request message at its own convenience. Since message passing systems do not wait for a response from the server function, these types of systems support asynchronous client/server interaction. However, the synchronous connection supported by RPC communication maintains a higher degree of reliability than message processing. In a message passing system, for example, it is possible that a message may never be received and neither the source, nor the receiver, would be aware of a problem.

RPCs have many other advantages, beyond the simple benefit of enhanced reliability. Some other advantages include the ability of RPCs to run on hosts having different operating systems, the ease of incorporation of RPC into various software products, and the ability to utilize unused CPU time at distant machines. However, RPC implementation has its drawbacks. RPC lacks flexibility and is often difficult to use with many servers. Message passing offers a greater degree of flexibility, along with an easier programming procedure for establishing asynchronous communication between processes in a networked environment. In summary, message passing lacks the reliability and standards that RPC provides and is limited in use (Korzeniowski 115).

RPCs vs. Local Procedure Calls

A remote procedure call appears extremely similar to a local procedure call--as intended. The difference between the two procedures is that in a local procedure call, the client process initiates a procedure in its own address space, whereas with RPC the server and client exist as two separate processes, usually on different machines (Comer 289). It is this separation of processes that allows the server function to reside on a different machine. Nevertheless, it is important to note that RPC can still be utilized when the client and the server execute on the same machine (Comer 306).

During normal implementation of RPC, the client process and the server process communicate to each other via two stubs (Levy 79), namely the client stub and the server stub. A stub is a communications interface that establishes the RPC protocol and determines how each message is constructed by the processes and interchanged between the two. The client process first consults its own stub to locate any remote processes that are required for program operation. Subsequently, the client makes the necessary requests of those processes. Meanwhile, the server (daemon) perpetually listens to the network, through the server stub, for any requests transmitted by clients. More specifically, one daemon, the Inetd, serves as a "grandfather" daemon for all other daemons. Inetd runs perpetually and starts other server daemons upon receipt of requests for server services. The server fulfills each request in succession, returning to its waiting state after completion of each request (Bloom 2).

At a more basic level, each server process is identified by a port (logical network communication channel) by which it establishes communications for client requests. When a server is initiated on a machine, the computer establishes an address (port) for server communications. This address, which is unique, is registered with the server machine's portmapper (Bloom 11-12).

The portmapper itself provides a crucial network service for all client/server communication. Its job is to keep track of all services that are available on a machine and their port addresses. Whenever a client requests a service from a particular machine, the client petitions the portmapper for the service. If the requested server exists, the portmapper establishes a communication channel between the client and the server. Even when a client and server reside on the same computer, the operation for establishing a link between the client and the server is the same; the network is still involved in the communication. The client still checks with its stub for the server's address, only in this case the address provided by the portmapper would correspond to the same machine. In effect, the request travels across the network only to return to its origin (Sun MicroSystems 36).

A Foundational Client/Server Demonstration

To facilitate the comprehension of programs upon approaching the tasks entailing RPC, it is helpful to begin with a straight-forward application. For this purpose, a client/server RPC demonstration has been developed wherein a client process passes an integer to a server that increments the integer and returns the updated value to the client. The files necessary to perform this, and any other, RPC application include a protocol definition file, client program, server program, and the stubs and header file generated by the RPC compiler after these are created.

Before programming the client and server processes, it is first necessary to create a protocol definition in the remote procedure call language (RPCL). A protocol definition is a file that describes both the list of data structures that will be passed between the client and server, and the function call required from the client in order to use the server's resources. The initial protocol definition file, *add_it.x*, can be referenced in Appendix A.

The critical elements in the protocol definition include a unique program number and version numbers within each program. In this example, the program is called ADDPROG and is assigned the unique number 0x20000002. It is imperative that this value be unique since it is used by the portmapper to identify the process (from poignant experience, the author can readily attest to the confusion that results from having multiple programs with the same program number). The version numbers are useful when updates warrant the need for a distinction between the original version, and the subsequent update (Bloomer 43). For example, the *add_it* protocol definition file contains one version with a simple function definition called ADD_NUM which receives and returns an integer. Another version could be defined (identified by the number 2) which passes a structure instead of an integer. RPCL is similar to the C language, though this simple example does not make this apparent; however, the protocol definition file for the author's application (Appendix B) demonstrates the similarity between declared constants, structure definitions, and additional functions.

Once the user has created the protocol definition file, a UNIX program, called RPCGEN, compiles the definition and produces several files. RPCGEN creates both the client and server stubs, as well as a header file that defines the RPC parameters that are included in both the server and client routines. After compilation of the RPC definition file, the next task entails developing the client and server code.

In the *add_it* example, the client code is labeled *add_it.c*. The code is written in C and contains familiar formats like *include* statements, variable declarations, etc. Additionally, the code includes features that are unique to RPC. First, the rpc/rpc.h library must be included along with the header file add_it.h which is produced by the compiler rpcgen. Next, a special pointer of type CLIENT is declared which points to a structure that contains information about the port and socket addresses (Bloomer 7). The value of that pointer is determined by the function clnt_create which establishes a connection between the server and client machines. This function requires the name of the host with which to establish a connection (can be the same), program name, version name, and the type of transport protocol (tcp or udp). If no connection can be established, the function clnt_pcreateerror is called to inform the user that no connection could be made to the host (Sun MicroSystems 45). Finally, the function that was declared in the protocol definition file (*add_it.x*) is called (the version number is appended to the function name by rpcgen). Passed to the function are the integer which will be manipulated and the CLIENT pointer which contains the communication information.

The last component of the *add_it* example is the server code, which is also coded in the C language. Again, the rpc/rpc.h library is included as well as the header file add_it.h. Noticeably different from the client code is the lack of a main declaration. In essence, the server code is simply a function declaration and can be considered as a function within the client code, only residing on a different machine. Furthermore, all communication between the client and the server is accomplished through pointers. Thus, the client passed the pointer to the integer *num* and the server returns a pointer to the new number which is, ironically, called *oldnum*. The descriptive "Hello People" statement was included to provide optional output to confirm that the server was responding (debugging tool).

Advanced RPCs

Once this simple RPC application that adds two integers over a network was completed, the level of difficulty was increased by working with strings and structures, until the author was versed enough in RPC protocol and application specification to begin work on the file migration application itself. It uses many of the fundamental concepts of the simpler RPC application, as well as additional, more complex concepts. A copy of the code is included in Appendices B through E. Appendix B is the protocol definition file, Appendix C contains the client code, Appendix D contains the server code, and Appendix E has all of the scripts written in UNIX that are used through system calls by the client and server code.

The application of file migration, of course, is not the only area where RPC can be employed effectively. There are many other applications which lend themselves to the advantages of RPC, including using RPC to calculate partial derivatives. In this application, the equations are partitioned and numerically solved on different computers. The fragmented solutions are then pooled to obtain the final result (Soto 25-27). Another application area encompasses using RPC as a tool for software applications dealing with real-time process control systems that are large and complex (Carpenter 901-902). Undoubtedly, the use of RPC will become more widespread and, as it does, we will see an increase in the number of applications in this area.

FILE MIGRATION

Many organizations have expanded significantly, in a physical and geographical sense, over the past few decades. This growth has been accelerated by the fast-paced nature of the advancing computing environment. Many organizations have noticed that it can be beneficial to modify the location of various frequently accessed company files from their current locations to others. By varying the location of files appropriately, communication costs and file transfer duration times, which typically result from locating and acquiring large files (such as company accounting histories or files containing multiple graphic bitmaps), can be minimized. Companies have often solved these problems by manually modifying the primary location of particular files, or by replicating the files to multiple locations.

A Case for Computer-integrated File Migration

For example, most world-wide corporations maintain personnel files for each and every member of their company. Regardless of each member's current location, it is imperative that the data contained within each file remains current. In order to maintain currency, the file must be updated continually as to reflect each person's current location, position, job status, possible job qualifications, personal preferences, etc. Most companies maintain each member's personal record at one specific site. The file is usually located closest to where that person normally conducts business (at some home-base location). Whenever their particular personnel file must be updated, their central file must be found and modified. Although this method may seem prudent, as long as the person remains at that specific location for an extended period of time and rarely deviates far from that location on business ventures, the reality of global business dictates that many personnel are frequently dispatched world-wide for extended periods of time at irregular intervals.

If a person is temporarily reassigned from one business site to another, for example, as the schedule of a sales representative could require, maintaining a file at a central location may be considered unwise. It makes sense, then, to have each company member's personal file "follow" them to their current business location, as required by their occupation. Many organizations have implemented this concept by manually moving personnel files from one location to another, as warranted. The process of moving this file is usually performed manually, but why bother when a computer system can perform the necessary file transfers, at close to optimal times, and minimize transferal and access costs concurrently? In most cases, the manual process of moving files is either cost-ineffective, inefficient, or subject to oversight errors.

A case for Computer-integrated File Replication

Related to the prior example, a person may perform certain business functions at two or more distant locations on a frequent and extended basis. In order to minimize communication costs, their personal file could be manually moved between these locations as their business functions require. However, this method may not be the most efficient and cost-effective one for determining the timing of file relocations. Routinely moving a file among two or more locations seems like a senseless task. If the employee regularly returns to a limited set of specific locations, having information specialists repeatedly relocate the member's personal file to each of these locations could result in unnecessary file transfer costs. If this person's file is only needed at each location for read-only applications, a simple file copy at each location would minimize the cost of access times and would require only one file transfer for each location. Any competent computer specialist could determine when a file should be copied to another location, but why employ one if the computer system can perform this function on its own? The system could apply a set of criteria and automatically determine when a file should be copied to another location. The possibility of automatically moving files to the optimal locations leads to the concept of file migration.

The Benefits of Computer-integrated File Migration

The two preceding examples illustrate the advantages of computer-integrated file migration. Although many methods can be employed to determine when a file should be either replicated or moved, there is little evidence to support that the file transfers themselves should be performed manually by the computer system monitors. A computer routine can be developed that determines the optimal distribution of each file and performs the necessary relocation(s) without human involvement.

The concept of file migration is, more or less, an automatic process. In this process, the organization pre-defines a specific set of criteria that must be met before a file is either relocated or replicated to another site. When the pre-specified criteria are met, the awaiting computer functions perform the desired file relocation and replication without any interaction by the computer system monitor. In this scenario, a centralized database would maintain the criteria as well as current information on the location of each file, the type of each file, and the types of accesses available to each file (e.g. if a file is read-only, write-only, or both). If the file location(s) have been modified, only a simple change to the database is required.

Although it may seem expensive to require each file query to consult the central database prior to any, and every, file access, the cost will usually be negligent, especially when compared to normal file transfers. For example, a personnel file that contains complex images (including dental exam x-rays, photographs, and other bitmaps) can often take hours to transfer error-free over a large distance, while a simple database call to another machine takes much less than a few seconds.

Analysis of Computer-integrated File Migration

So, as far as the added communication overhead is concerned, file migration adds little to the overall cost. Actually, this extra overhead cost is required from all personnel database systems, since any distributed system requires the existence of, and access to, some type of database to determine a particular file's location. The real savings from the file migration concept comes from the efficiency gained by the elimination of human interaction at the file transfer level. The transferal/replication criteria are only specified at a single point, with periodic updates, eliminating the necessity of human decision-makers to determine the location of each file at each point in time. The system itself takes care of the implementation issue.

APPLICATIONS FOR FILE MIGRATION

In order to demonstrate RPC, three different, but related, file migration applications were chosen with regard to unit personnel records. For instance, in the Air Force, currently all records are kept on one database in one location. While this may seem like the simplest way to maintain a database, it might be more cost-effective to distribute the files and move them electronically. This is the basic idea behind the applications. While each option is a slightly different scenario with respect to migrating personnel records, all three applications have an underlying goal as well as a communal set of code.

This shared objective is to create an application that will maintain databases of the actual personnel records along with a database that knows the location of each record. Furthermore, each application will display a set of statistics regarding the location and number of accesses of a particular file from a particular location. This enables an individual to keep track of where the file is used and requested most often. Finally, based on some predetermined set of rules (here is where the different options are derived) the file is transferred to a different location and a message appears telling the user that the file has been transferred to a new location. The transfer is also updated in the database that tracks the locations of the files.

As mentioned before, the file transfer is governed by a set of rules. Here is where each application finds its identity. The first option deals with transferring a file based on a location that is different from the current location of the file. The second option transfers a file based on the percentage of requests made. The final option transfers a file at regularly scheduled times to given locations. There exist complex algorithms that dictate when a file should be migrated based on a system's resource utilization, file sizes, and the number of write and read accesses to those files (Hac 1459); however, because the emphasis of this paper is on remote procedure calls rather than file migration techniques, these algorithms are beyond the scope of this project.

Shared Application Features

Though each of the three applications address a different decision rule, they all share certain attributes, processes, and implementation techniques. Since the authors developed a generic set of code around which the applications are built, each application shares a number of the same variables. Furthermore, many of the functions within the server code are identical since the same file information is being processed in each application.

Of special interest is the use of the rdbpt server as a client of the fts server. In other words, the same process can function as a server and a client! The server process need only make a request of another server to become a client, too. This feature in the code required extensive debugging for two reasons: first, the location in the code of the second server call was important to the server's operation and second, the structure of the compilation file was altered significantly in order to integrate the additional stubs into the original configuration.

Finally, each application implements C's capability to make UNIX system calls. This capability allows the integration of UNIX scripts which are triggered by C with a simple system call to the name of the script. In this way, the various tasks of the applications are performed in the most appropriate environment. For example, it is easier to work with files using UNIX rather than C since the code resides on UNIX machines. The UNIX system calls are easily identified throughout the code.

An Application for Timed File Migration

In real-world applications for file migration, most files are relocated when specific criteria are met. In other situations, however, the criteria of basing file relocation due to the number and location of accesses from various hosts is inadequate or inefficient.

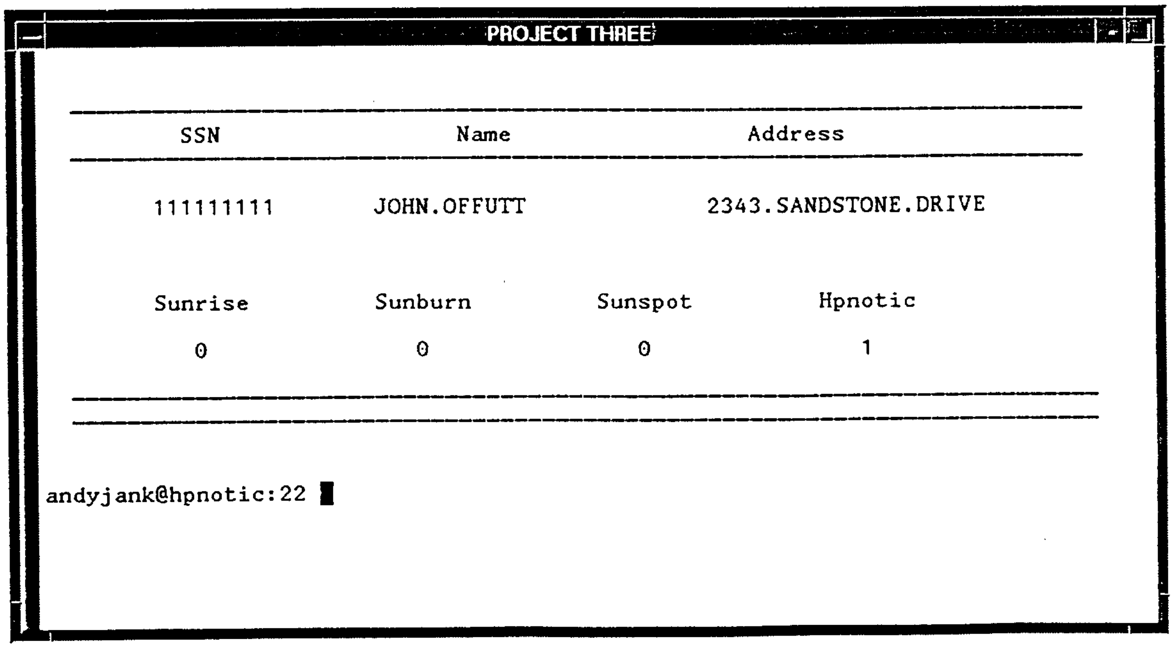
For example, a global corporation may have a specific set of general files that must remain current on a company-wide basis. Let us also suppose that these files are only accessed at specific times of the workday (e.g. files that each company location must access at 9 a.m. to start the workday, and again at 5 p.m. to report changes to the files). Although it may seem easier to the casual observer to simply let a set of criteria determine the optimal file locations, it should be noted that 9 a.m. and 5 p.m. are different for each time zone in which the company operates. Why not just automatically relocate the file on the hour as each of the companies in a specific time zone are expected to begin accesses, rather than force the system to determine the proper timing?

This application demonstrates one technique for addressing this type of problem by invoking a computer-clocking routine to automatically move the files at certain times to specific locations. A process is located at the central database that performs this function. Although the files are automatically moved at the desired times to the new time zone locations, other processes can be involved, concurrently, to determine the optimal location for the files within each time zone.

The application scenario typifies an application for a continental US corporation. In our example, we assume that the one server exists in each of the time zones, and at specific intervals, a subset of three "organizationally important" files are dynamically moved across the US from east to west as the time zone shifts would require. The remaining set of files are kept at their current locations and can be relocated either manually, or by some concurrently operating optimization process.

Running the Application

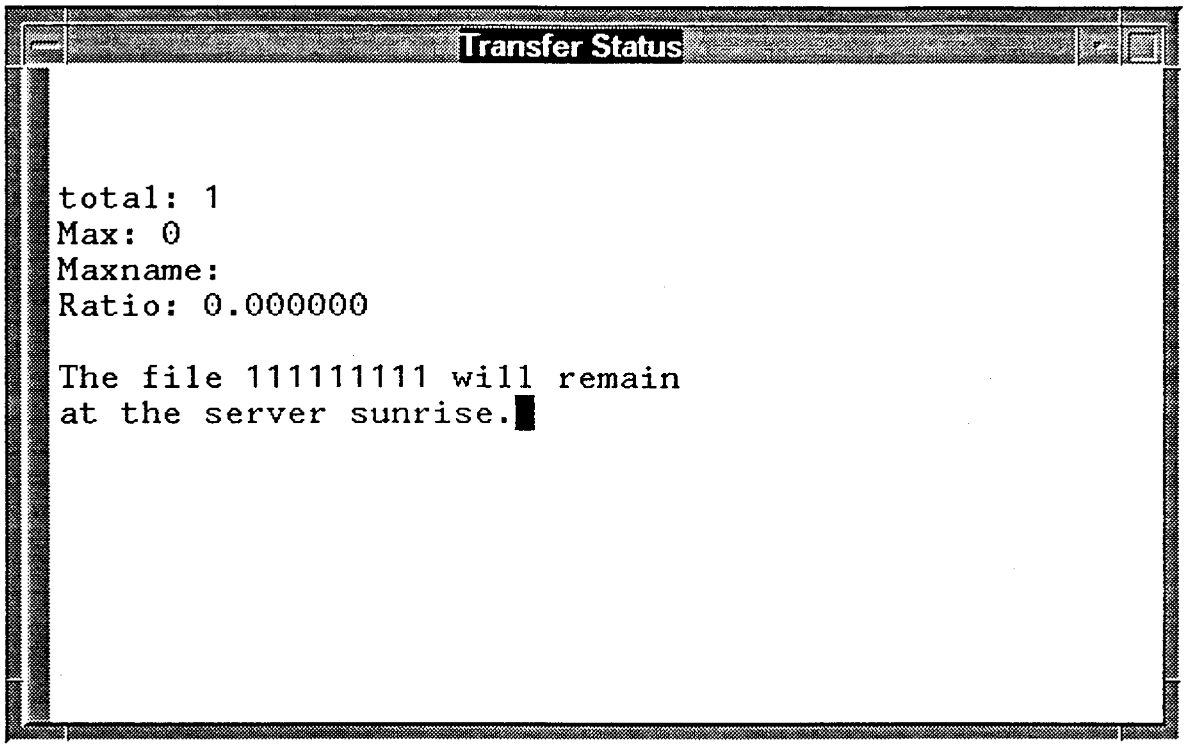
To initiate the application, the user may either click on the appropriate icon, or simply execute the RUN_IT script (Appendix F). The script itself merely removes all of the temporary files created by the last run of the application, re-initializes the data to the beginning state, and executes the application in the proper application windows. In order to facilitate the compilation process, RUN_IT also executes the MAKEFILE script (Appendix F) which automates the process of integrating the program stubs generated by RPCGEN (from the two files RDBPT.X and FTS.X, Appendices B and D, respectively) into a cohesive application.



```
      SSN                 Name                  Address

    111111111        JOHN.OFFUTT          2343.SANDSTONE.DRIVE


     Sunrise           Sunburn         Sunspot          Hpnotic
        0                 0               0                1


andyjank@hpnotic:22
```

Figure 1.

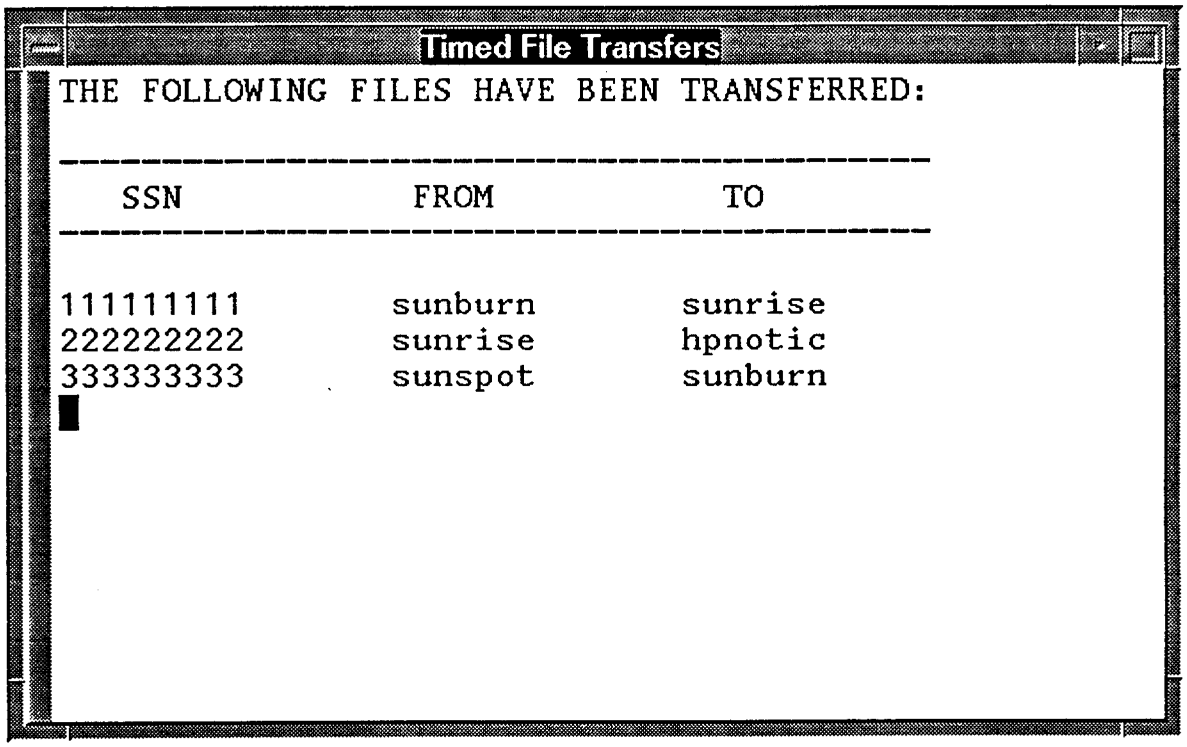
When the application is executed, the user will initially see three windows. The main (active) window, titled "Project Three", displays the user's prompt and the results of each database query. The results include not only the desired record, but also the current number of accesses to that file from each server. To make a database query, the user would input a command of the form > rdbpt hpnotic 111111111 (indicating a request for the personnel file "111111111" from the central database located at server "hpnotic"). After executing the query, the database will be updated and the results will be displayed in the main window (see Figure 1).



```
total: 1
Max: 0
Maxname:
Ratio: 0.000000

The file 111111111 will remain
at the server sunrise.
```

Figure 2.

As the query is processed, the second window, titled "Transfer Status", will display the status of the requested file (see Figure 2). The status indicates whether or not the file has been dynamically relocated from its current server to another. Incidentally, the window also displays a set of diagnostic statistics that keep track of the total number of accesses for that file. For a more complete description of these values and their implications, please refer to the report on migration based on heuristics by Eric DeLange.



```
THE FOLLOWING FILES HAVE BEEN TRANSFERRED:

------------------------------------------
   SSN            FROM           TO
------------------------------------------

111111111       sunburn       sunrise
222222222       sunrise       hpnotic
333333333       sunspot       sunburn
```

Figure 3.

At one-minute intervals (simulating a one-hour time-elapse), a third window, titled "Timed File Transfers", will appear for ten seconds (see Figure 3). This window displays the timed file migration of the three "organizationally important" files across time zones. The window reports what files have been migrated and their new locations. Future accesses to these files via the main window will prove that the files have, indeed, changed locations (as indicated in the Transfer Status window).

As long as both of the two server processes are active, queries to the database can be completed from any of the valid server locations. It should also be noted that no actual file relocations are performed by the application, since it is beyond the scope of the project.

Discussion of RPC Operation and Programming Code

At a first glance, the application may seem relatively simple, but in reality, only an examination of the underlying programming code can reveal the complexity of the application.

The application itself consists of one client process, RDBPT.C, and two server processes, RDBPT_SVC_PROC.C and FTS_SVC_PROC.C. When a query is made to the database, in this case, for the file "111111111" from central database server "hpnotic", the RDBPT.C client code is executed (Appendix B). Initially, the client uses two UNIX scripts, FIND_HNAME and SSN_IDENT (Appendix F), to convert portions of the query into the proper variable representation, and subsequently establishes a connection with the RDBPT server. The client then uses this connection to query the central database for the actual location of the requested file and waits for the result.

At this time, the awaiting server (RDBPT_SVC_PROC.C) receives the request to execute the function *ssn_key_1* and return the current location and updated statistics for that file. the function executes the sub-functions *read_dbase* and *write_dbase* to get the current file location, update the file's access statistics, and write the new information to the database. The server returns the information to the client and awaits another request.

Once the waiting client receives the actual location of the file, it makes another request to the server to return the file contents. To limit the scope of this project, this request is also made to the RDBPT server at the same host location. However, the application is designed to accommodate distributed systems, such that an identical server residing on another host could be queried, given that a few modifications must be made to the RDBPT code. The server executes the *get_rec_1* function to retrieve the record.

The *get_rec_1* function establishes a connection with the file transfer server, FTS_SVC_PROC.C (Appendix D), that resides on the host owning the desired information file. In this case, the file is located on the same host as the central database, but, in an actual distributed system application, the file could be located on any known host. The RDBPT server now acts as a client, passes the request for file information, and waits for an answer.

The file transfer server receives the request and executes function *trans_1* to return the file information. Although the file transfer is not actually performed, this is the server that would discharge that duty in a fully developed distributed application. Instead, the function simply displays a message to the "Transfer Status" window, indicating the location of the desired file and whether or not the file has been moved.

The file transfer server returns a status code to the RDBPT server, which, in turn, returns the record information to the initial client. The client then displays the file contents and the file's access statistics in the "Project Three" window.

Discussion of Timed File Transfers

The above process continues as long as both servers are active and awaiting requests from various clients. However, an additional set of processes are also running concurrently. At timed intervals, as dictated by the AT_FILE (Appendix E), a certain set of "important" files (111111111, 222222222, 333333333) are migrated from host to host as a group, simulating the hourly time zone changes. At each timed execution, the script TIME_WINDOW (Appendix E) displays the "Timed File Transfers" window and automatically moves the files. The TIME_WINDOW script executes another script, TIMECOP (Appendix E), that actually updates the central database with the new locations of each file. Again, the file transfers are not actually performed, but a fully developed distributed application would include a function to perform the necessary file transfers. The script ends by presenting the file transfer information.

The effect of this event-driven procedure can be inspected by querying the database for one of the "important" files both before and after a timed file transfer has occurred. The "Transfer Status" window will indicate the new location of the file.

Notice that in this application, the clock time is an *event* that triggers file migration. The application can be classified as an example of an "active" or "event driven" file migration system. Although clock time is the only event affecting migration in this application, other events could be integrated into the system (such as current processor loads, remaining disk capacity, status of other network hosts, the arrival of a message signaling specific file transferals, and so forth). Accordingly, the application is consistent with a growing body of active system literature that is attracting considerable attention in information technology.

RPC FRONT ENDS WITH HP-UX

The front-end display is an important addition to any application, especially for users. Applications that use RPC technology typically produce a large number of complicated, and almost indecipherable, files. Additionally, in order to run any RPC application, the server application must already be waiting for a request at the server end, before the client can be executed. The typical user does not know, or care, about any of the implementation issues. One attractive quality of client/server computing is that the complicated aspects of computer integration and application are transparent to the user. Therefore, it is important that any user interface that utilizes RPC functions facilitates running the program, and all of its additional files, without involving the end-user in the details of its underlying operation.

The HP-UX Workspace Environment

The HP-UX machines provide several different viewing sessions that offer applications and tools that can be accessed by simply clicking a pointing device (ex. a mouse) on an icon. Two of these sessions are HP VUE and HP VUE lite. The basic differences between the two are that the HP VUE lite session has a front panel with different features and different methods for determining how the view manager interprets the information provided by the user. These panels are typically displayed whenever the user logs on, much how Microsoft Windows provides a set of panels, each of which contains a set of program icons. Users can customize their own panels, but only within the limits set by the System Administrator, who determines what type of options are available to each class of user.

Developing a Control

The application was developed within the HP VUE session. The HP VUE session panel consists of a top and bottom row. In order to modify these rows several changes must be performed: an icon must be constructed, an action that will be associated with that icon must be defined, the action must then be linked to the icon to define a control, and finally the control (the icon with an underlying action) must be added to the front panel by editing the ".vuemrc" file.

Building Icons

Creating an icon is relatively easy for anyone who has a creative mind and likes to draw. The IconEditor is specifically designed for this purpose; it provides the user with a drawing screen and tools to construct various icons. Once the icon is finished, the constructor need only save it for future use. In order to have the icon associated with an application, an action must be specified and attached to the icon.

Creating Actions

Creating an action is accomplished by simply using the CreateAction icon in the HP-UX General Toolbox. The user inputs the action name, a command line to initiate the application, and a type of window to display the action results within. An action can only be specified once; if future modifications must be made, a new icon-action definition must be specified. Once the icon and action associated with it have been created, the newly defined control can be added to the panel.

Adding a Control to the HP-UX VUE Session

The ".vuemrc" file must be edited to include the new control. The ".vuemrc" file contains all the control definitions, as well as the names for the controls that reside in the top and bottom rows of the panel. These controls can take on different actions, as well as combinations of behaviors including: push button, drop zone, file monitor, client window, and toggle button.

To include a new control to the ".vuemrc" file, several steps must be taken. Initially, the icon controls are placed in the appropriate box and are then defined. The box placement describes in what order the various controls will be displayed. The word "CONTROL" along with the name for a new control are included among the other control definitions for a desired box (top or bottom) and position within the box. The type of control must also be defined. In the case of the application, a button "TYPE" control is used to initiate the application. For any control addition, the "PUSH_ACTION" for the button must also be specified. The "PUSH_ACTION" can either be an action similar to the one created above, or an executable command. An action similar to the one created above is incorporated into the author's RPC application. Finally, an "IMAGE" must be specified for each control. The image is simply the icon name. Little more is involved with modifying the view configuration of the HP-UX machines to include any, and all, new applications to the window manager.

Restarting the Workspace

In order to incorporate the new controls to the window manager, all one needs to do is restart the workspace manager. The new control definitions will take effect and function in all future uses of the start-up window manager workspace. By using icons and associating them with actions it is possible to execute the RPC application in an environment that not only preserves the benefits of client/server computing, but also retains the invaluable element of end-user transparency by isolating the user from the underlying complications involved with RPC technology.

CONCLUSIONS

Though the author's experience in the area of RPC is limited, by researching the subject and working on a project that incorporates its capabilities, the usefulness of RPC in communicating in a client/server environment was recognized. Moreover, even though the project was somewhat limited, the complexity involved in implementing client/server technology is readily discerned. Of course, the project was not completed without learning something, nor were the possible applications of RPC in the realm of file migration exhausted.

Lessons Learned

As with any project, there is no substitute for the lessons one learns from experience. This project is no exception. First and foremost, it cannot be stressed enough how important it is that the program number defined in the protocol definition file be unique. If there are two programs with identical values, a clear communication channel cannot be established between a client and **one** server since, according to a client's stub, two server's will exist for the same purpose.

Another important issue in dealing with RPC is passing parameters. It is necessary that all variables be passed as pointers for RPC to work properly. Additionally, the value that is to be returned from the server function must be declared as a static variable.

Future Endeavors

In order to keep the scope of this project within reason, the three file migration applications were approached from a simulation perspective. That is, no physical migration of files actually occurs. When a file is said to have been moved from one host to another, the only thing that has changed is the database file that would track the migration of files. Obviously, this leaves open the possibility of expanding the project by physically migrating the files. Furthermore, additional applications can be developed such as maintaining duplicated files at separate sites (read-write and read-read access are important in this area). Along with this, more research can be done in adequate rules that define at what point a file is to be moved or duplicated. This does not comprise an exhaustive list of supplementary application areas to the project and the reader is free to explore their own possibilities.

BIBLIOGRAPHY

Bloomer, John. Power Programming with RPC. Sebastopol: O'Reilly and Associates, Inc., 1991.

Carpenter, B. E. and R. Cailliau. "Experience with Remote Procedure Calls in a Real-time Control System." Software--Practice and Experience. Vol. 14. September 1984, p. 901-07.

Comer, Douglas E. and David L. Stevens. Internetworking with TCP/IP. Vol 3. Inglewood Cliffs: Prentice Hall, Inc., 1993.

Curry, David A. Using C on the UNIX System. Sebastopol: O'Reilly and Associates, Inc., 1989.

Hac, Anna. "A Distributed Algorithm for Performance Improvement Through File Replication, File Migration, and Process Migration." IEEE Transactions on Software Engineering. Vol 15, No 2. November 1989, pp. 1459-1470.

Hahn, Harley. A Student's Guide to UNIX. New York: McGraw-Hill, 1993.

Hewlett-Packard Company. HP Visual User Environment 3.0 User's Guide. Hewlett-Packard Company, 1992.

Kernighan, Brian W. and Dennis M. Ritchie. The C Programming Language. 2nd ed. Murray Hill: AT&T Bell Laboratories, 1988.

Korzeniowski, Paul. "Make Way for Data." Byte. June 1993, pp. 113-115.

Levy, Henry M. and Ewan D. Tempero. "Modules, Objects and Distributed Programming: Issues in RPC and Remote Object Invocation." Software--Practice and Experience. Vol. 21. January 1991, pp. 77-90.

Soto, J. L. Cruz, M. C. Calzada Canalejo and M. Marin Beltran. "Parallelization of Differential Problems by Partitioning Method (Synchronized Algorithm)." Computers and Mathematics with Applications. July 1993, pp. 25-31.

Sun MicroSystems. Network Programming Guide. Sun MicroSystems, Inc., 1990.

Waite, Mitchell and Stephen Prata. New C Primer Plus. Carmel: Sams Publishing, 1993.

APPENDICES

Appendix A

Contents:

```
/*  ADD_IT.X     */
/*  THIS FILE IS USED BY RPCGEN TO PRODUCE THE HEADER FILE ADD_IT.H */



program ADDPROG {
        version ADDVERS {
                int ADD_NUM(int) = 1;
        } = 1;
} = 0x20009300;
```

```
/*      ADD_IT.C     */
/* THIS IS THE CLIENT PROCEDURE */

#include <stdio.h>
#include <ctype.h>
#include <rpc /rpc.h>
#include "add_it.h"

main()
{
CLIENT *c1;
int num;
char *hostname[20];

printf("\nPlease enter the remote host name: ");
gets(hostname);

printf("\n\nPlease enter the number to increment: ");
scanf("%d", &num);

/* THIS IS THE CLIENT HANDLE THAT ESTABLISHES THE CONNECTION */
/* BETWEEN THE CLIENT AND THE SERVER */

if (!(c1 = clnt_create(hostname, ADDPROG, ADDVERS, "tcp")))
{
   clnt_pcreateerror(hostname);
   exit(1);
}
                                   /* HERE IS THE PROCEDURE CALL */

printf("\n\nThe new number is %d\n", *(add_num_1(&num, c1)));

}
```

```
/* ADD_IT_SVC_PROC.C */
/* THIS IS THE SERVICE PROCEDURE */



#include <stdio.h>
#include <string.h>
#include <rpc /rpc.h>
#include "add_it.h"

int *add_num_1(oldnum)
int *oldnum;
/* THIS FUNCTION ADDS 21 TO THE INPUT NUMBER AND THEN RETURNS THE NUMBER */
{
   printf("\nHello People");
   *(oldnum) += 21;
   return oldnum;
}
```

Appendix B

Contents:

```
/*                                                                              *
/*                    *** PROJECT THREE ****                                    *
/*                                                                              *
/*                                       RDBPT.X                                *
/*                                                                              *
/*                    This program is used by RPCGEN to produce the header file *
/*                              for the RDBPT RPC.                              *
/*                                                                              *



const MAX_REC_LEN = 255;                /* max personal record length
const SSN_SIZE = 9;                     /* size of Social Security Number
const NAME_SIZE = 80;                   /* size of person's name
const ADDR_SIZE = 80;                   /* size of person's address

const HOST_SIZE = 255;                  /* size of host computer


/*                    Defines the sturcture of each personal record:            *
/*                    SSN, name, and address                                    *
/*                                                                              *

struct pers_rec
{
        string    ssn<SSN_SIZE>;
        string    name<NAME_SIZE>;
        string    address<ADDR_SIZE>;
};


/*      Defines the structure of the database record for each file:            *
/*                    ssn (filename), current location, four counters to measure *
/*                    file accesses from each server.                           *
/*                                                                              *

struct dbase_rec
{
        string    ssn<SSN_SIZE>;
        string    loc<HOST_SIZE>;
        int       sunrise;
        int       sunburn;
        int       sunspot;
        int       hpnotic;
};


/*                    Defines the structure of an information record about a specific *
/*                    file, including the SSN (filename), the local host from the file* /
/*                    has been requested, and where the file is actually located *
/*                                                                              *


struct inputrec
{
        string ssn<SSN_SIZE>;
        string lc_host<HOST_SIZE>;
        string h_name<HOST_SIZE>;
};

```

```
program RDBPT_PROG
{
        version RDBPT_VERS
        {
                dbase_rec SSN_KEY(inputrec) = 1;
                pers_rec GET_REC(inputrec) = 2;
        } = 1;
} = 0x20009306;
```

```
/*                                                                              */
/*                         *** project 3 ***                                    */
/*                                                                              */
/*                                   RDBPT.C                                    */
/*                                                                              */
/*                  This is the client code for the RDBPT service.              */
/*                                                                              */
/*                                                                              */

#include <stdio.h>
#include <string.h>
#include <rpc /rpc.h>
#include "rdbpt.h"
#include <stdlib.h>

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

{
        CLIENT          *cl;            /* client   handle          */
        dbase_rec       *d_record;   /* retrieved db record
        pers_rec        *p_record;      /* retrieved personal record   */
        inputrec        inrec;          /* information record (ARGV'S) */
        FILE            *c_fp = NULL;   /* FP for hostname conversion  */
        FILE            *ssn_p = NULL;  /* FP for ssn conversion       */
        char            HSTNAME[HOST_SIZE];
                                          /* temp variable for hostname */

/*                 Assign filename and hostname to variables for use              *


        inrec.ssn = argv[2];
        inrec.h_name = argv[1];


/*                 Convert hostname to proper representation                      *


        system ("find_hname");
        c_fp = fopen("hname.txt","r");
        fscanf (c_fp, "%s", HSTNAME);
        fclose (c_fp);
        inrec.lc_host = HSTNAME;


/*                 Converts SSN filename to proper representation                 *


        ssn_p = fopen("ssn.txt", "w");
        fprintf(ssn_p, "%s", inrec.ssn);
        fclose(ssn_p);
        system("ssn_ident");

        p_record                = (char *)      malloc(sizeof(pers_rec));
        d_record                = (char *)      malloc(sizeof(dbase_rec));


/*                 Provides error usage message                                   *


        if ((argc != 3))
        {
```

```
                        fprintf(stderr, "\nUsage: %s local.server SSN\n", argv[0]);
                        exit(1);
                }

/*                       Establishes connection to the server (RDBPT) process              *

                if (!(cl = clnt_create(argv[1], RDBPT_PROG,
                                           RDBPT_VERS, "tcp")))
                {
                        clnt_pcreateerror(argv[1]);
                        exit(1);
                }

/*                       Retrieve file location and access information                     *

                d_record = ssn_key_1 (&inrec, cl);

/*               Retrieve actual file                                                     *

                p_record = get_rec_1 (&inrec, cl);

                system ("clear");                                  /* Clear screen

/*                       print out the contents of the file, the current file loction     *
/*                       and the current file access statistics                           *

                printf ("\n\n   ————————————————————————————————————————");
                printf ("————————————————————————————————————\n");
                printf ("\t   SSN\t\t          Name\t\t     Address\n");
                printf ("   ————————————————————————————————————————");
                printf ("————————————————————————————————————\n");

                printf ("\n\t%s\t%s\t\t%s\n\n\n\n", p_record->ssn, p_record->name,
                                           p_record->address);

                printf ("\tSunrise\t\tSunburn\t\tSunspot\t\tHpnotic\n");
                printf ("\n\t      %d\t\t       %d\t\t      %d\t\t      %d\n\n",
                        d_record->sunrise, d_record->sunburn, d_record->sunspot,
                        d_record->hpnotic);

                printf ("   ————————————————————————————————————————");
                printf ("————————————————————————————————————\n");
                printf ("   ————————————————————————————————————————");
                printf ("————————————————————————————————————\n\n\n");
}
```

Appendix C

Contents:

Rdbpt_svc_proc.c

```
/*                                                          */
/*        ***  project 3  ***                               */
/*                                                          */
/*                              RDBPT_SVC_PROC.C            */
/*                                                          */
/*                This is the program that provides services to the */
/*                          RDBPT clients.                  */
/*                                                          */

#include <stdio.h>
#include <string.h>
#include <rpc /rpc.h>
#include "rdbpt.h"
#include "fts.h"
#include <ctype.h>

FILE              *fp1 = NULL;      /* FP for accessing central DB
FILE              *fp2 = NULL;      /* FP for retrieving personnel  record
FILE              *ssn_p = NULL;    /* FP for opening temporary files      */
static pers_rec   *p_rec = NULL;    /* pointer for personnel record        */
static dbase_rec *d_rec = NULL;   /* pointer for database record
char              *maxname[15];     /* name of location with the max accesses */
char              *temp_loc[15];    /* temp var for old file location

/*                         READ_DBASE                                  *
/*                This function allocates internal memory and reads the central   *
/*       database, retrieving the statistics and host information about  *
/*                the desired personnel file                          *
/*                                                                     *

int read_dbase(inrec)
        inputrec               *inrec;

{
        if (!d_rec)
        {
                d_rec = (dbase_rec *) malloc(sizeof(dbase_rec));
                d_rec->ssn = (char *) malloc(sizeof(SSN_SIZE));
                d_rec->loc = (char *) malloc(sizeof(HOST_SIZE));
                d_rec->sunrise = (int) malloc(sizeof(int));
                d_rec->sunburn = (int) malloc(sizeof(int));
                d_rec->sunspot = (int) malloc(sizeof(int));
                d_rec->hpnotic = (int) malloc(sizeof(int));
        }

        if (fscanf(fp1, "%s %s %d %d %d %d",
                d_rec->ssn, d_rec->loc,
                &d_rec->sunrise, &d_rec->sunburn, &d_rec->sunspot,
                &d_rec->hpnotic) != 6)
                        return (0);

        return (1);

}
```

```
/*                              WRITE_DBASE
/*
/*                   This function modifies the statistics counter and file location    */
/*                   information and updates the applicable central database records    */


int write_dbase(inrec)
        inputrec                *inrec;
{
        int                     total, maximum;
        float                   ratio;

/*                              increment appropriate counter if same as the            *
/*                              local host requesting service.                          *


        if (strcmp(inrec->lc_host, "sunrise") == 0)
                d_rec->sunrise += 1;
        if (strcmp(inrec->lc_host, "sunburn") == 0)
                d_rec->sunburn += 1;
        if (strcmp(inrec->lc_host, "sunspot") == 0)
                d_rec->sunspot += 1;
        if (strcmp(inrec->lc_host, "hpnotic") == 0)
                d_rec->hpnotic += 1;


/*                   calculates the total number of accesses from all locations         *


        total = (d_rec->sunrise + d_rec->sunburn + d_rec->sunspot +
                 d_rec->hpnotic);




        system("clear");
        printf ("\n\n\ntotal: %d", total);
        if (total > 5)                          /* Only calcs max if total is over */
                maximum = max();                  /* some given number
        printf("\nMax: %d", maximum);
        printf("\nMaxname: %s", maxname);
        ratio = (float) maximum /total;      /*calculates the comparison       */
        printf ("\nRatio: %f\n",ratio);    /* ratio                            */


        strcpy(temp_loc, d_rec->loc);

/*   Evaluation criteria for file transfer                                              *

        if (ratio > 0.2)
        {
          strcpy (d_rec->loc, maxname);
          d_rec->sunrise = 0;
          d_rec->sunburn = 0;
          d_rec->sunspot = 0;
          d_rec->hpnotic = 0;
        }
```

```
/* Writes the new database changes to the disk files                                  *

                fprintf(ssn_p, "%s %s %d %d %d %d\n",
                        d_rec->ssn, d_rec->loc,
                        d_rec->sunrise, d_rec->sunburn, d_rec->sunspot,
                        d_rec->hpnotic);


                return (1);
}

/*                      This function finds the location with the max number of accesses   * /

int max()
{
                int maximum;

                maximum = d_rec->sunrise;
                strcpy(maxname, "sunrise");

                if (d_rec->sunburn > maximum)
                {
                        maximum = d_rec->sunburn;
                        strcpy(maxname, "sunburn");
                }

                if (d_rec->sunspot > maximum)
                {
                        maximum = d_rec->sunspot;
                        strcpy(maxname, "sunspot");
                }

                if (d_rec->hpnotic > maximum)
                {
                        maximum = d_rec->hpnotic;
                        strcpy(maxname, "hpnotic");
                }

                return maximum;

}



/*                    READ_PERS_REC                                                    *
/*                                                                                   * /
/*                      This function reads the desired personnel record into  the     *
/*                      structure p_rec.                                               *


int read_pers_rec()

{
                if (!p_rec)
                {
                        p_rec = (pers_rec *) malloc(sizeof(pers_rec));
                        p_rec->ssn = (char *) malloc(sizeof(SSN_SIZE));
                        p_rec->name = (char *) malloc(sizeof(NAME_SIZE));
                        p_rec->address = (char *) malloc(sizeof(ADDR_SIZE));
                }
```

```
        if (fscanf(fp2, "%s %s %s",
                p_rec->ssn, p_rec->name, p_rec->address) != 3)
                        return (0);
        return (1);
}


/*                              SSN_KEY_1                                      *
/*                                                                             *
/*                   This function is called remotely by an established client *
/*      The client sends the local host location and personnel filename * /
/*                   and this server returns the actual location of the personnel *
/*      file, as well as the accumulated statistical information about   * /
/*                   that particualar file                                     *


dbase_rec  *ssn_key_1(input_rec)
        inputrec                *input_rec;
{
        char *DFILE;

        DFILE               = (char *) calloc(MAX_REC_LEN+1, sizeof(char));
        strncpy(DFILE, "filedb.txt",MAX_REC_LEN);


        if (!(fp1 = fopen (DFILE, "r+")))
                return ((dbase_rec *) NULL);

        while (read_dbase(input_rec))
                if (!strcmp(d_rec->ssn, input_rec->ssn))
                        break;

        ssn_p = fopen("tempdb2", "w");

        if feof (fp1)
        {
                fclose (fp1);
                return ((dbase_rec *) NULL);
        }



        write_dbase(input_rec);
        fclose (fp1);
        fclose(ssn_p);
        system("/users/andyjank/project3/changefile");

        return ((dbase_rec *) d_rec);
}


/*                              GET_REC_1                                      *
/*                                                                             *
/*                      This function is called remotely by an established client          * /
/*          The client sends the host location and personnel filename and     * /
/*                      this server returns the personnel file.                                          *

pers_rec *get_rec_1(input_rec)
          inputrec                    *input_rec;

{
          CLIENT      *c2;
          info_rec info;
          char *number[20];
          char *PFILE;
          pers_rec                    *error;

          PFILE                       = (char *) calloc(MAX_REC_LEN+1, sizeof(char));
          strncpy(PFILE, "D.",MAX_REC_LEN);

          strcat(PFILE, input_rec->ssn);


          if (!(fp2 = fopen (PFILE, "r")))
                    return ((pers_rec *) NULL);

          while (read_pers_rec())
                    if (!(strcmp(p_rec->ssn, input_rec->ssn)))
                              break;

          if feof (fp2)
          {
                    fclose (fp2);
                    return ((pers_rec *) NULL);
          }
          fclose (fp2);

/*  This will call another server that would actually perform the file  * /
/*  transfer in a fully implemented application                                              *

info.oldloc = temp_loc;
info.newloc = d_rec->loc;
info.filename = d_rec->ssn;


if (!(c2 = clnt_create("hpnotic",FTSPROG, FTSVERS, "tcp")))
{
   clnt_pcreateerror("hpnotic");
   exit(1);
}

trans_1(&info, c2);


return ((pers_rec *) p_rec);


}
```

Appendix D

Contents:

```
/*                    *** project 3 ***                                          */
/*                                                                               */
/*                                   FTS.X                                       */
/*                                                                               */
/*                          This file is compiled by RPCgen to produce the       */
/*                    header file for our file transfer server.                  */
/*                                                                               */

const HOST_SIZE = 255;
const F_NAME_SIZE = 255;

                    /* Structure that contains the original location of the
                    /* file, the filename, and the desired new file location*/

struct info_rec
{
        string    oldloc<HOST_SIZE>;
        string    newloc<HOST_SIZE>;
        string    filename<F_NAME_SIZE>;
};


program FTSPROG
{
        version FTSVERS
        {
                int TRANS(info_rec) = 1;
        } = 1;
} = 0x20009305;
```

```
/*                  *** project 3 ***                                   *
/*                                                                      *
/*                          FTS_SVC_PROC.C                              *
/*                                                                      *
/*                  This file is compiled by RPCgen to produce the      *
/*              file transfer server executable.  The executable itself *
/*              does not actually perform the file transfer to the new  *
/*              server; it only produces a message informing the user   *
/*              that this is where the actual transfer would take place. *
/*                                                                      *


#include <stdio.h>
#include <string.h>
#include <rpc /rpc.h>
#include "fts.h"




int *trans_1(h_info)

info_rec            *h_info;

{
        static int num;
        num = 1;

        if (strcmp(h_info->oldloc,h_info->newloc) == 0)
          printf("\nThe file %s will remain \nat the server %s.",
                  h_info->filename,h_info->oldloc);
        else
          printf("\nThe file %s has been \nmoved from %s to %s.",
                 h_info->filename, h_info->oldloc, h_info->newloc);

   return (&num);
}
```

Appendix E

Contents:

```
#! /bin /csh -f

#                              TIMECOP
#
#          This UNIX script is used to identify those files which will be
#          moved based on our third application of time dependency.
#          The files are isolated and their current locations are
#          identified at which point they can be moved to the appropriate
#          host and then the information is outputted to the screen.
#

#
#          These are the files that will be moved every time period
#

grep 111111111 filedb.txt >! file_temp
grep 222222222 filedb.txt >> file_temp
grep 333333333 filedb.txt >> file_temp

#
#          Identify what host the file is currently at and move
#          it to the host which needs it next according to some
#          established time sequence.
#

grep -q hpnotic file_temp
if ($status == 0) then
   grep hpnotic file_temp | sed s /hpnotic /sunspot / >> file_temp1
endif

grep -q sunrise file_temp
if ($status == 0) then
   grep sunrise file_temp | sed s /sunrise /hpnotic / >> file_temp1
endif

grep -q sunburn file_temp
if ($status == 0) then
   grep sunburn file_temp | sed s /sunburn /sunrise / >> file_temp1
endif

grep -q sunspot file_temp
if ($status == 0) then
   grep sunspot file_temp | sed s /sunspot /sunburn / >> file_temp1
endif

#
#          Sort the files and join them so that it can be printed
#

sort file_temp >! file1
sort file_temp1 >! file2

join -j1 1 -j2 1 -o 1.1 1.2 2.2 file1 file2 >! display_temp_file

sed s Λ  Λ \ \ \ \ \ \  /g display_temp_file >! display_file

egrep -v "(111111111|222222222|333333333)" filedb.txt >> file_temp1

cat file_temp1 >! filedb.txt
sort filedb.txt >! file3
cat file3 >! filedb.txt

#
```

```
#           Remove all temporary files
#

rm file_temp*
rm file1 file2 file3 display_temp_file

#
#           Standardize the screen output
#

echo "THE FOLLOWING FILES HAVE BEEN TRANSFERRED:"
echo ""
echo "————————————————————————————————————————"
echo "    SSN              FROM              TO          "
echo "————————————————————————————————————————"
echo ""
cat display_file
sleep 10
```

```
#! /bin /csh  -f

#
#                                     TIME_WINDOW
#
#       This short UNIX script creates a window so that the results
#       from the script timecop can be output to the screen.
#

xterm -title "Timed File Transfers" -geometry 53x18+644+5 -e /users /andyjank /project3 /timecop
```

```
#! /bin /csh  -f

#
#                                   At_File
#
#         This file conntrols the timed executions of application 3
#

at -f time_window now
at -f time_window now + 1 minutes
at -f time_window now + 2 minutes
at -f time_window now + 3 minutes
```

Appendix F

Contents:

```
#!  /bin /csh  –f

#                         CHANGEFILE:                          #
#                                                              #
#          This UNIX script updates the database file.         #
#                                                              #

cat tempdb2 >> tempdb
cat tempdb >! filedb.txt
```

```
#! /bin /csh -f

#                         FIND_HNAME:                          #
#                                                              #
#     This UNIX script finds the hostname of the client.       #

echo `hostname` >! hname.txt
```

```
#! /bin /csh

#                         MAKEFILE:                              #
#                                                                #
#         This file compiles the rdbpt RPC and form the needed   #
#         object files.                                          #

makedata
rpcgen  rdbpt.x
rpcgen  fts.x

cc -c -o rdbpt.o  rdbpt.c
cc -c  rdbpt_clnt.c
cc -c  rdbpt_xdr.c
cc -o  rdbpt  rdbpt.o  rdbpt_clnt.o  rdbpt_xdr.o

cc -c -o  rdbpt_svc_proc.o  rdbpt_svc_proc.c
cc -c  rdbpt_svc.c

cc -c  fts_xdr.c
cc -c -o  fts_svc_proc.o  fts_svc_proc.c
cc -c  fts_svc.c

cc -o  fts_svc  fts_svc_proc.o  fts_svc.o  fts_xdr.o
cc -o  rdbpt_svc  rdbpt_svc_proc.o  rdbpt_svc.o  rdbpt_xdr.o  fts_svc_proc.o
```

```
#!  /bin /csh  -f

#                        SSN_IDENT:                          #
#                                                            #
#         This UNIX script takes the input SSN and creates a file #
#         that includes all non-matches of that SSN.         #

grep -v `cat ssn.txt`  /users /andyjank /project3 /filedb.txt > tempdb
```

```
#! /bin /csh -f

#                               RUN_IT:                                 #
#                                                                       #
#          This UNIX script starts the two servers and positions        #
#          their corresponding windows appropriately.                   #


cd  /users /andyjank /project3
 /users /andyjank /project3 /clean
 /users /andyjank /project3 /makefile

xterm -title "Transfer Status" -geometry 53x18+5+5 -e  /users /andyjank /project3 /rdbpt_svc&

 /users /andyjank /project3 /fts_svc&

 /users /andyjank /project3 /at_file

xterm -title "PROJECT THREE" -geometry 80x24+200+400
```