A comparative study of structured and un-structured remote data access in distributed computing systems

Tang, Wai Chung

University of Durham

School of Engineering and Applied Science
(Computing Science)

A comparative study of structured and un-structured remote data access in distributed computing systems

WAI CHUNG TANG

A thesis submitted for the Degree of Master of Science

Volume 2

Contents

Preface

This is the second volume of the thesis which consists of the last three chapter of the main thesis, a set of appendices, a glossary section and followed by a reference section.

Having provided the required background knowledge for the project in the first five chapters, a distributed system based on the programming languages PS-algol (introduced at the end of chapter five) and C is described in chapter six. Chapter seven examines the tradeoffs of the two data access models by presenting two detailed applications on the implemented system. Finally, chapter eight concludes the work of this thesis.

There are in total six appendices which intend to give additional information about the thesis. The glossary section presents the definitions of the important computing terms or phrases that have been come across during the course of the work.

Finally, the thesis is ended by a section which lists out all the books, manuals and journals that have been referenced.

Chapter Six -- Design and implementation issues of a distributed system

Over the past five chapters, the fundamental principles of distributed computing systems (DCSs), abstract data types (ADTs) and remote procedure call (RPC) techniques have been addressed. These concepts form the basic philosophy of a distributed system named as DCSUNIX - Distributed Computing System on UNIX, which will be described in this chapter.

This chapter is divided into four sections: the first section presents the main motivations for DCSUNIX, the second section depicts the overall structure of the system, the third section identifies the basic hardware and software components of the system and finally the last section concludes this chapter by discussing the overall achievements of DCSUNIX.

6.1. Objectives

DCSUNIX is a small scale experimental distributed computing system based on the persistent programming language PS-algol (see Appendix B) and the language C in conjunction with the RPC facitities available on SUN[1] Unix[2] 4.2 BSD operating system.

The reasons of choosing the language PS-algol have been discussed in chapter five. On the other hand, the operating system Unix is employed for the development of DCSUNIX due to the facts that:

(a) Unix is proven multi-user, multi-tasking environment,

(b) Unix supports powerful and sophisticated RPC programming facilities which are mostly written in the language C,

- - - - - - - - - - - - - - - - - -

[1] SUN is a trademark of SUN Microsystems Inc.

[2] Unix is a trademark of the Bell laboratories.

(c) It is fairly easy to install PS-algol on Unix,

(d) The portability of software written on Unix systems.

The aims of DCSUNIX are as follows:

(a) Provide a distributed environment for the investigation of the nature of structured and un-structured remote data access strategies as mentioned in chapter one.

(b) Determine the tradeoffs of the two access methods in (a) by performing some experiments on DCSUNIX.

(c) As an example of the construction of distributed application programs using the notions of ADTs and RPC.

N.B. The two terms ADTs and objects will be used interchangeably for the rest of this chapter.

6.2. Overview

Figure 6.1 below shows the basic structure of the DCSUNIX system:

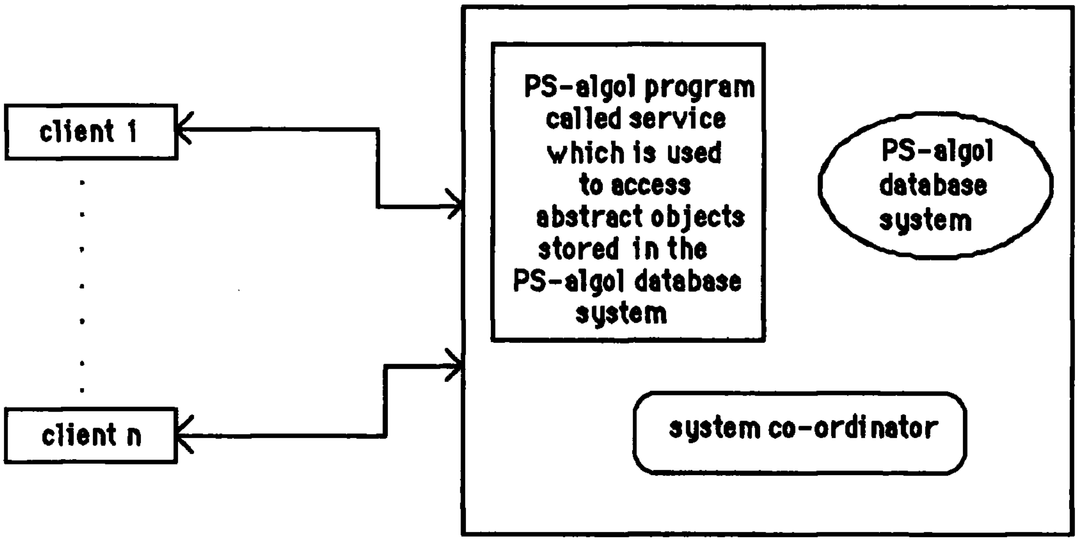


Figure 6.1 The overall design of the DCSUNIX system

As illustrated by the above diagram, DCSUNIX is designed with the expectation of supporting multi-user data processing. This system is made up of two major parts: a server and a set of n clients, where n is the total number of authorized users. Both the server and clients are *nodes* of a computer network.

The server is responsible for the following tasks:

(a) To provide a repository for mounting the PS-algol persistent database system in which objects (represented as ADTs) are stored.

(b) To provide a PS-algol module (program) named **service** which contains all the applications that the DCSUNIX system can handle.

(c) To initiate a C module called the **system co-ordinator** which co-ordinates the processing and responses to the remote client processes (or termed 'user process') by executing the appropriate application in the module **service**.

On the other hand, the activities occurred in a client process when it makes a request for a service, which will involve the access of an object stored in the PS-algol database system, from the server are:

(a) Communication channels for data transfer are established between the client and the server by making remote procedure calls.

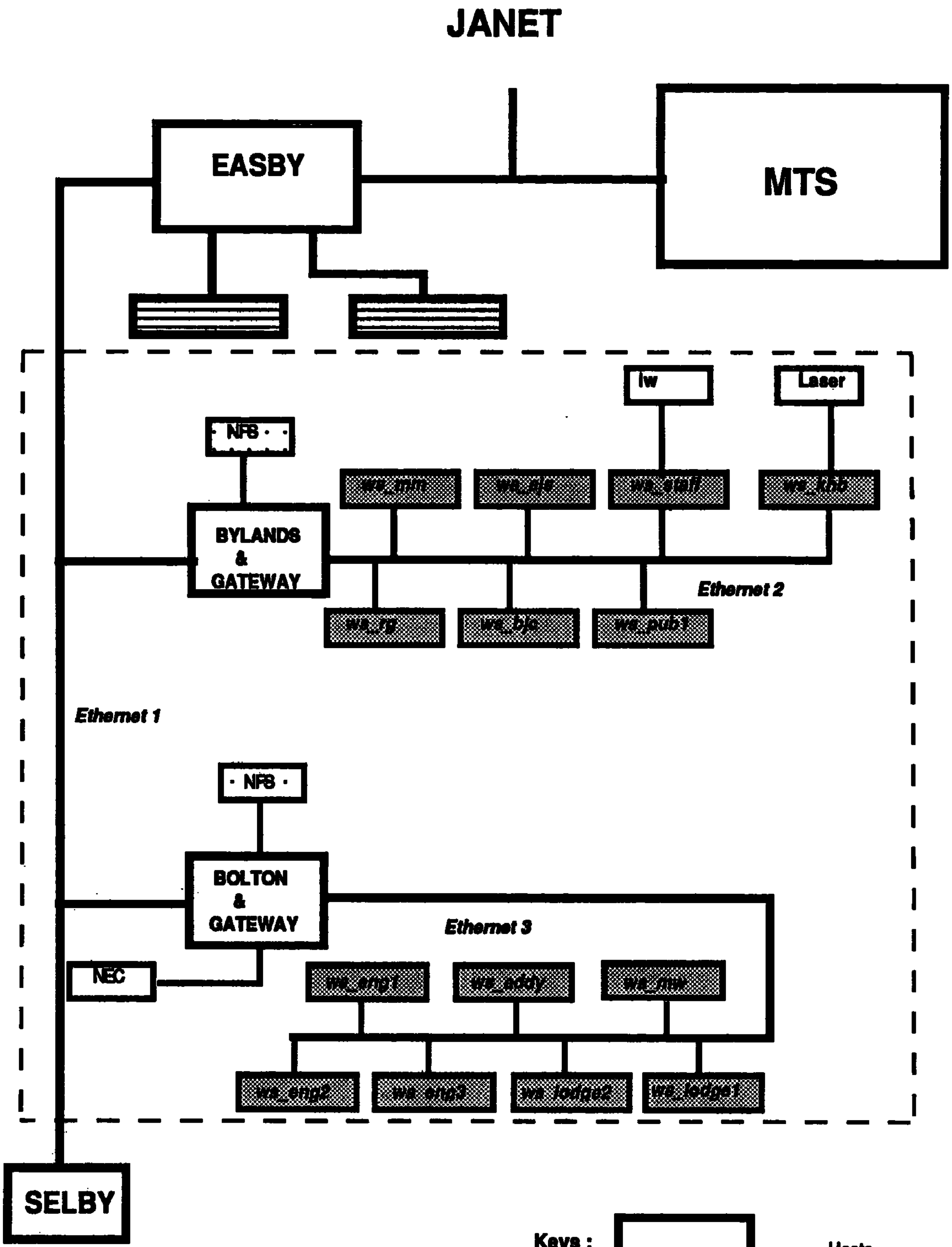
(b) The client process supplies the appropriate parameters by which the server can accomplish the requested service.

6.3. Consitutents of the DCSUNIX system

6.3.1. Hardware

The dashed area of Figure 6.2 depicts the computing environment available for the development of DCSUNIX. There are in total 14 SUN 3/50 workstations and two local hosts - known as bylands and bolton. The main difference between a workstation and a host is that the latter possesses a **gateway server** which converts messages and protocols between two networks.

The Ethernet communication system [76] is the principal means for communication. An Ethernet is a broadcast, packet-switched, digital network that can connect up to 256 computers, separated by as far as a kilometer, with a 3M bits/sec channel. Although Ethernet is an efficient low-level packet transport mechanism which gives its best efforts to delivering packets, it is not error-free. even when transmitted without an error detected by the sender, a packet may still not reach its destination without error; thus packets are delivered only with high probability.



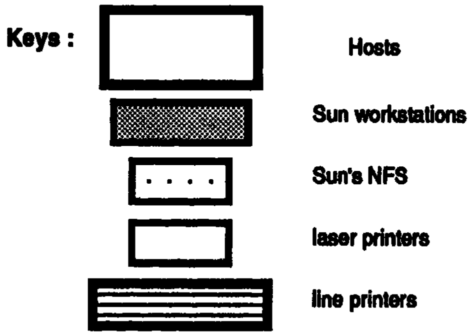


Figure 6.2 Implementation Environment of DCSUNIX

In order to reduce the possibility of overloading, the Ethernet system is divided into three parts. Each of them is responsible for a specific portion of the network as indicated in Figure 6.2 (Ethernet 1, 2 and 3). This arrangement also provides certain degree of fault-tolerance because the network can still work when one of the hosts or workstations crash due to some kind of hardware failure. Besides, it is more flexible in the sense that different types of networks can be attached to each other via the appropriate gateway servers.

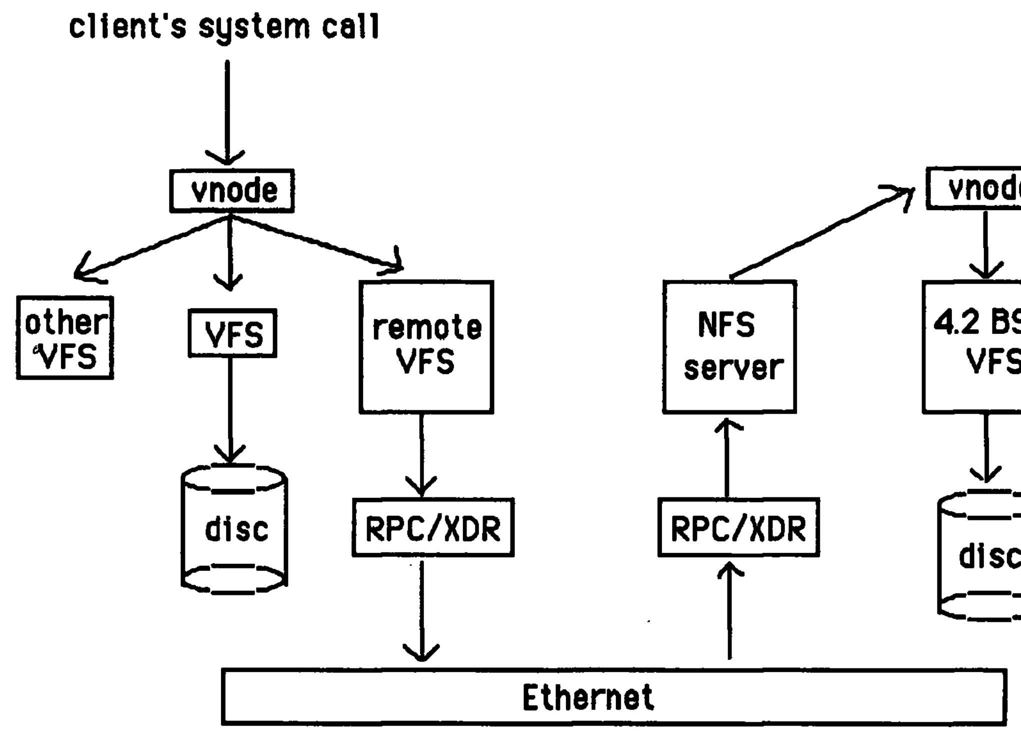
Theoretically, any one of the machines (workstations and hosts) can act as the server or a client at a particular instant since they all employ the same operating system Unix 4.2 BSD and they also share the same *filestore* (not shown in Figure 6.2) via SUN's Network File System (NFS) as described below. However, no client process can ever be generated in the machine that has been designated as the server.

Architecture of SUN's NFS:

In general, SUN's NFS provides a facility for sharing files in a heterogenous environment of machines, operating systems and networks. Sharing is accomplished by mounting a remote file system, then reading and writing files in place. The NFS is open-ended which means it can be freely connected to other systems. So, NFS is not a distributed operating system, but rather, an interface to allow a variety of machines and operating systems to play the role of a client or a server despite the fact that NFS is composed of a modified Unix *kernel*: a set of library routines and a collection of utility commands.

Traditionally, a Unix file system is consisted of directories and files. Each file has a corresponding **inode** (index node) containing adminstrative information about the file such as location, size, ownership, permissions and access times. Inodes are assigned unique numbers within a file system, but a file on one file system could have the same number as a file on another file system. This is a serious problem in a network environment because remote file systems need to be mounted dynamically and numbering conflicts would cause confusion. To solve this problem, SUN has designed the virtual file system (VFS) based on **vnodes** which are generalized implementation of inodes that are unique across file systems.

The following diagram shows the flow of a request from a client to a collection of file systems:



So, above the VFS interface, the operating system deals in vnodes; below this interface, the file system may or may not implement inodes. The VFS interface can connect the operating system to other file systems, eg. MS-DOS or 4.2 BSD. A local VFS connects to file system data on a local device. However, the remote VFS defines and implements the NFS interface using remote procedure call (RPC) mechanisms. RPC allows communication with remote services as if they are called locally. The RPC protocols are described using the eXternal Data Representation (XDR) package which permits a machine-independent representation and definition of high-level protocols on the network (see Appendix B for more information about RPC and XDR).

Referring back to the flow diagram, in case of access through a local VFS, requests are directed to file system data on devices connected to the client machines; in the case of access through a remote VFS, the request is passed through the RPC/XDR layer onto the network. In the current implementation, SUN uses the

User Datagram Protocol (UDP) and Ethernet for interprocess communications. On the server side, requests are passed through the RPC/XDR layer to an NFS server; the server uses vnodes to access one of its local VFSs and services the request. This path is retraced to return results.

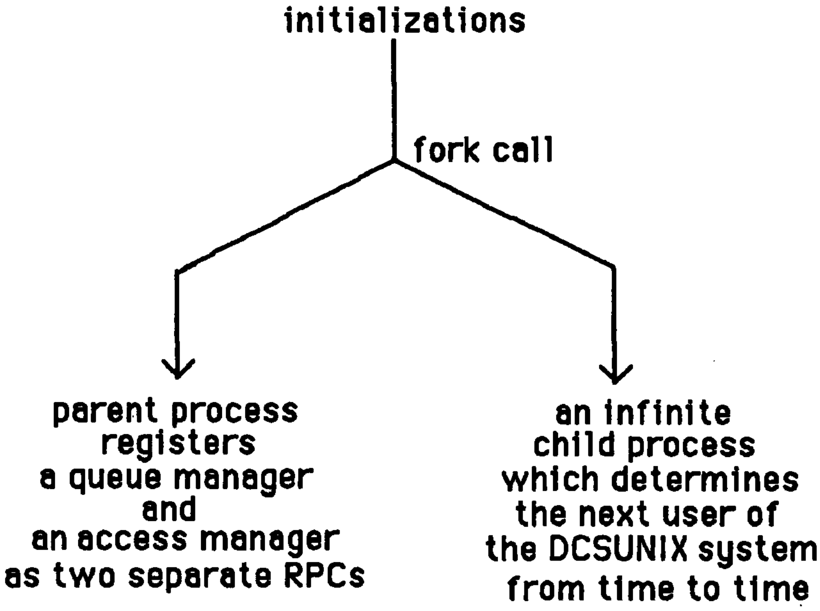
The final point about SUN's NFS is that the NFS interface is defined such that a server can be **stateless.** This means that a server does not have to remember any information (apart from its local files information) between transactions. The major advantage of stateless server is robustness in the face of client, server or network failures. If a client process crashes, it is not necessary for a server to take any actions to continue normal operations. If a server or the network crashes, it is only necessary that clients continue to attempt to complete NFS operations until the server or the network is fixed. This robustness is especially important in a complex network of heterogenous systems, many of which may be running untested systems and/or may be rebooted without warning.

6.3.2. Software

According to the design of DCSUNIX shown in Figure 6.1, the following number of software modules are identified. Note that this section will involve some of the Unix system and C library function calls. Details of these routines can be found in [77] if required.

6.3.2.1. On the server side

The chief responsibility of the server is to provide client processes a way of accessing objects stored in the PS-algol database system. To achieve this goal, an un-terminated C module (once it has started) known as the **system co-ordinator** is established. The functions carried out by this module are summarized in the following paradigm:



As illustrated by the above diagram, the only task of the parent process is to register the two managers as RPCs and then it will go into an infinite loop waiting to service requests.

The child process of the system co-ordinator, the queue manager and the access manager are closely-related processes. By making use of the Unix **pipe** facilities, the first two processes set up a First_In_First_Out (FIFO) queue for DCSUNIX. Before proceeding any further, the concept of a Unix pipe will be explained briefly. In Unix, a pipe is an interprocess communication channel which is created via the invokation of Unix's **pipe** system call:

```
pipe(fildes)
int  fildes[2];
```

Two file descriptors are returned: **fildes[0]** and **fildes[1]**. The former is opened for reading only whilst the latter is for writing only. When the pipe is written using the descriptor fildes[1], up to 4096 bytes of data can be buffered before the writing process is blocked. On the other hand, the read only file descriptor fildes[0] accesses the data written to fildes[1] on a FIFO basis. After the pipe is set up, two co-operating processes can pass data through the pipe with the **read** and write system calls. However, for **read** calls on any empty pipe (i.e. no buffered data) with only one end (all write file descriptors are closed) an end-of-file will be returned. Similarly, an error signal will be generated if a write on a pipe with only one end is attempted.

Relationship of the queue manager and the child process:

In order to implement the queue mentioned earlier, two Unix pipes are required. They are called the **host_fildes** and **procnb_fildes** respectively. Whenever a user request is received, which may be happening simultaneously, by means of the RPC mechanisms (see Appendix C), the queue manager will place the caller process's identification (ID) details into the respective pipes so that these information can be retrieved at a later stage as explained shortly. The following pseudo-code describes the functions of the queue manager more precisely.

```
queue manager:
    begin
      accepts ID information from a client process;
      write(host_fildes[1], client process's local machine name);
      write(procnb_fildes[1], client process's local process number);
    end;
```

In constrast to the task of the queue manager, an infinite child process created by the system co-ordinator (using the Unix **fork** system call) extracts ID information from the queue so as to play the role of an entrance guard of the entire system. The whole algorithm works as follows:

Since the queue is a FIFO one and the identification details have been entered to the two pipes by the queue manager in the same order as they are received, so the frist value of each pipes must correspond to the same caller process. Using these two values, known as **caller_hostname** and **pid**, as the parameters, the child process informs the caller process at the remote site that it is its turn to use the system by calling a procedure inside its parent process (the system co-ordinator) named **restart_proc**. The basic mechanism used in this procedure is a combination of the Unix's **rsh, kill** and **system** calls [77]. The routine **system** is a standard C library function which issues shell commands via a string of characters as if they are entered through the keyboard. In this particular case, the required string will be of the form:

system("rsh caller_hostname kill -19 pid")

The effect is to connect to the specified remote machine using the remote shell command - **rsh**, and then a restart signal (**kill** -19) is sent to the local process with ID pid. The only problem in constructing such a string concerns with the last argument pid; it is an integer whereas all the others can be expressed as character strings. This gives rise to another procedure called **itos** defined in the parent process which takes pid and an empty character array as parameters. By evaluating the value of each digits of pid, it converts them into the corresponding ASCII characters before copying them into the empty array as the result. Having done this, the final **system** call can be issued.

Up to this stage, the child process needs to be suspended, via a local **system** call with **kill** -17 and its own process number as arguments, so that only one process at a time could have control over DCSUNIX. In other words, consistency can be maintained. After the caller process has completed its task, the child process will be re-activated to serve another process using the queue as before. The overall structure of the child process can be summarized as:

<u>system co-ordinator's child process</u>:

```
while true do
  begin
    caller_hostname := read(host_fildes[0]);
    pid := read(procnb_fildes[0]);
    restart_proc(caller_hostname, pid);
    child_ID := get the child process's own ID;
    system("kill -17 child_ID");
  end;
```

Access manager:

Although the role of this process is quite different from the other processes discussed so far, the connection between this process and the child process of the system co-ordinator is vital to the whole system. This process is initiated (using the RPC mechanism again) by the client process which has just acquired control of the DCSUNIX system through the child process. Figure 6.3 portrayed all the activities associated with thre access manager.

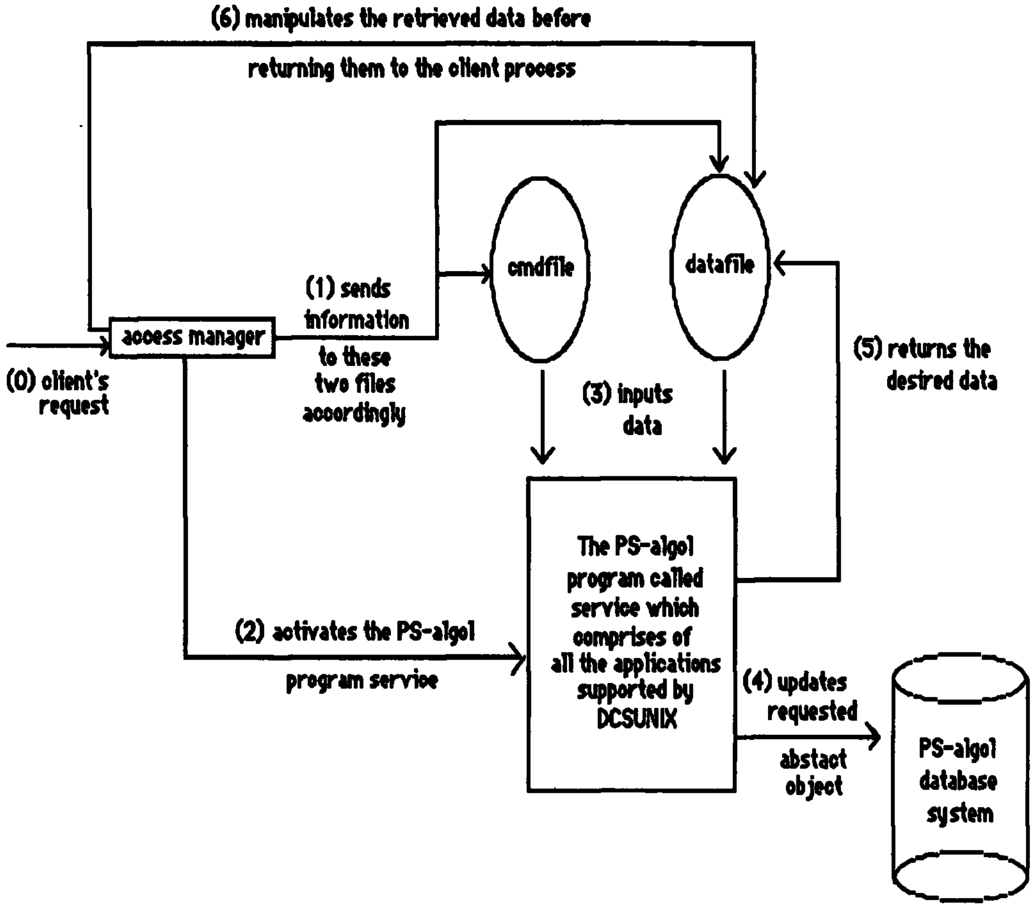


Figure 6.3 Responsibilities of the access manager

The numbers appeared in the figure indicates the ordering of events occurred during the execution of the access manager process. Accompanying the access manager are two Unix files, **cmdfile** and **datafile**, and a PS-algol program named **service**. The first Unix file cmdfile is designated as a repository for maintaining information that defines uniquely the command of a specific user service and the location where the operation of the service will be undertaken. This file has four components: the name of the service required, the name of the operation to be performed, the name of a specific database and the entry name of this database. The last two components of cmdfile identifies the abstract object that has been installed to the specified database by means of the technique described in section 5.2.4.2. of chapter five. Note that all the four components of cmdfile are supplied by the access manager as indicated in Figure 6.3.

On the other hand, the file datafile serves either one of the following functions:

(a) For read mode, that means information is read from the abstract object specified by the last two items of cmdfile, this file is used to keep the return status of the client's request which indicates whether or not the request has been performed successfully, followed by the result of the request issued by the program **service** as described very shortly.

(b) For write mode, this file is required in the following two separate stages. Before the executing the client's request, it keeps all the update data of the abstract object stored in the specified database location. After the execution, it is used to keep the return status of the request. However, an extra item will be added by the program **service** if the request is failed as described shortly.

Another closely-related module to the access manager is the PS-algol program service. The following pseudo-code outlines the general structure of this module:

```
Declaration part:
   constants   maxsize, read.mode, write.mode, call.ok, call.fail;
   datatype1   data1[maxsize];
   datatype2   data2[maxsize];
        :
   datatypeN   dataN[maxsize];
   integer     item;
   string      msg;
Main program body:
   open(cmdfile, read.mode);
   application := readln(cmdfile);
   case application of
        service1: begin
                       command := readln(cmdfile);
                       dbname := readln(cmdfile);
                       entry := readln(cmdfile);
                       close(cmdfile);
                       if command is in write mode then
                       begin
                          open(datafile, read.mode);
                          item := 0;
                          while not eof(datafile) do
                          begin
                             item := item +1;
                             data1[item] := read(datafile);
                          end;
                          close(datafile);
                          open(dbname, write.mode);
                          execute command(dbname, entry, data1);
                          if the execution succeeded then
                          begin
                             open(datafile, write.mode);
                             write(datafile, call.ok);
                             close(datafile);
                          end
                          else begin
                                  msg:= generates an error message;
                                  open(datafile, write.mode);
                                  write(datafile, call.fail);
                                  write(datafile, msg);
                                  close(datafile);
                               end;
                          close(dbname);
                       end
```

```
                    else begin
                            open(dbname, read.mode);
                            data1 := execute command(dbname, entry);
                            if the execution succeeded then
                            begin
                               open(datafile, write.mode);
                               write(datafile, call.ok);
                               write(datafile, data1);
                               close(datafile);
                            end
                            else begin
                                    msg:=generates an error message;
                                    open(datafile, write.mode);
                                    write(datafile, call.fail);
                                    write(datafile, msg);
                                    close(datafile);
                                 end;
                            close(dbname);
                         end
     service2 to N : similar to service1 except data1 is replaced
                     by data2, data3 and so on;
end;
```

It should be noticed that program **service** has contained all the underlying data structures of the abstract objects stored in the entire PS-algol database system: definitions datatype1 to datatypeN (currently N is 2). Thus, the correct data type can always be applied to the corresponding application provided that the same data structure is adopted by the access manager when supplying the input data. As the result, various kinds of data structures, which have been classified as the structured and un-structured data types in this thesis, can be passed to the DCSUNIX system for testings as demonstrated in the next chapter.

After the execution of **service**, the access manager returns a record to the client process (step 6 in Figure 6.3) which contains:

(a) the process number of the system co-ordinator's child process such that the client process can re-activate the child process upon termination;

(b) an integer to indicate the return status of the client's request: 1 means success and 0 means fail;

(c) a variant record which keeps the returned data of the client's request if it is succeeded, otherwise, an error message is stored in the record. Also, bearing in mind that the former exists only when the request is in the read mode whilst the latter may exist in both read and write modes in case of an error.

6.3.2.2. On the calling side

Only a client process (sometimes referred as the caller process) is present which initiates a specific request for accessing an object (represented as an abstract data type) stored in the server's PS-algol database system. Note that the client process must have knowledge about the following information in advance:

(a) the name of the server machine,

(b) the name of its own machine,

(c) the identities of the two RPCs, queue manager and access manager of the system co-ordinator, such as their procedure numbers, version numbers and so forth (see Appendix C),

(d) the representation of the data structure(s) that is to be sent to or received from the server.

Generally, a client process is structured as below (in pseudo-code):

```
declaration:
    import the relevant header files from the system library;
    constants  server_name, client_name;
    data types  info=record
                        caller : a string of characters;
                        pid : integer;
                     end;
```

```
        data=record
                user-defined data structures;
             end;
        statuskinds=(ok, fail);
        result= record
                  childproc_ID : integer;
                  case status:statuskinds of
                      ok: (values : data);
                      fail: (msg: a string of characters);
                  end;

var  process_ID : info;
     input_data : data;
     returned_data : result;
     str : a string of characters;
     success : boolean;

procedure  itos(var str : a string of characters; intnb : integer);
     begin
        converts intnb into the corresponding ASCII string
        and then stores the result in variable str;
     end;

procedure  callrpc_queue_manager( process_ID : info;
                                  var success : boolean);
   begin
        calls the RPC routine queue_manager of the system
        co-ordinator at the server site with process_ID as parameter;
        if the call is ok then success := true else success :=false;
   end;

procedure  callrpc_access_manager( input_data : data;
                                   var returned_data : result);
   begin
        calls the RPC routine access_manager of the system
        co-ordinator at the server site with input_data as argument;
        if the call is ok then
        begin
             stores the result of the RPC in
             the record variable return_data;
        end
   end;
```

<u>Main program body</u>:

```
   process_ID.caller := client_name;
   process_ID.pid := the process number of this module;
   callrpc_queue_manager(process_ID, success);
```

```
if success then
begin
    itos(str, process_ID.pid);
    sends a stop signal to this module itself using the variable str
    and then waiting for its turn to use the DCSUNIX system;
    input_data := construct all the required information
                  for the client's request;
    callrpc_access_manager(input_data, returned_data, success);
    if the call is ok then
    begin
        manipulates the data stored in the variant
        record field returned_data.values;
        sends a re-start signal to the system co-ordinator's child
        process via the number stored in returned_data.childproc_ID;
    end
    else prints the error message stored in the
         variant record field returned_data.msg;
end
else generates an error message;
```

Several points are worth mentioning about the above algorithm:

(a) The two constants **server_name** and **client_name** define the names of the server machine and the client process's local machine respectively. The former is solely used by the two callrpc procedures as explained in point (d) below whereas the latter is only required in the main program body of the client process to initiate the whole transaction.

(b) The user-defined data type, **data**, is just adopted as a place holder, as in the type definition **result**, which signifies the fact that the definition of this data type is free to the user provided that the server is capable of dealing with that particular data type. Another data type called **info** is used to keep the local machine name and process number of the client process so as to put this process into the server's FIFO queue. Finally, the data type **result** stores the information returned after the completion of the client's request.

(c) Procedures **itos** has exactly the same code and responsibility as the one possessed by the system co-ordinator of the server.

(d) The two procedures, **callrpc_queue_manager** and **callrpc_access_manager**, are responsible for invoking the two RPCs queue_manager and access_manager of the system co-ordinator respectively. The implementations of these two procedures are very similar to the sample given in section 4.1 of Appendix C except that the number of parameters and the timeout values are different. Procedure callrpc_access_manager possesses an additional record variable called **returned_data** which holds the information returned from the server after the call. The timeout values of these two procedures are defined in such a way that the former allocates to each client process a maximum of one try (about 1 second of normal time) per transaction whilst the latter allocates a maximum of three tries (about 30 seconds of normal time each try) per transaction. As a result, the system can respond to more than one user process simultaneously with the aid of the server's FIFO queue and also sufficient time has been reserved for allowing the server to complete each user request. Furthermore, both these two procedures have to use the server's machine name (i.e. constant **server_name**) to create a communication channel for the two RPCs.

(e) The boolean variable success is merely used to indicate the return status of each RPCs - true or false.

(f) Finally, the following Unix **system** calls are required to generate the stop and re-start signals mentioned in the algorithm:

system("kill -17 pid")

and

system("rsh server_name kill -19 pid")

where pid is the process number of the target process.

6.3.2.3. Summary

So far, all the major software components of the multi-user distributed system DCSUNIX have been described. Apart from the PS-algol program service, all the other modules are implemented as C codes. The purpose of this sub-section is to present an overall picture of the established system as portrayed by Figure 6.4:

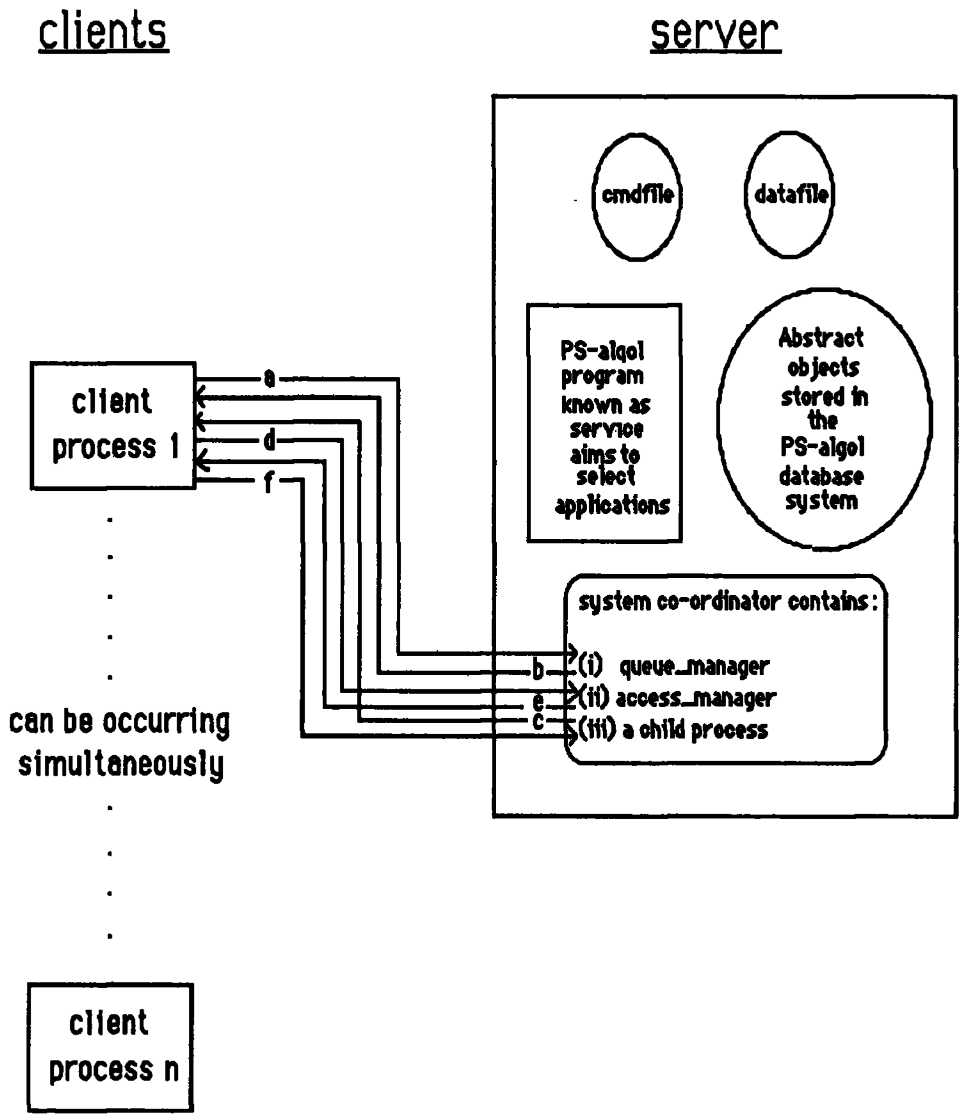


Figure 6.4 Relationships between the server and the client processes

Referring to the above figure, the following events are taken place:

Stage (a)

A client process invokes the first RPC routine, queue_manager of the system co-ordinator, via the RPC mechanism as described in Appendix C, with its local machine name and its process number as parameters.

Stage (b)

The queue_manager puts the client process's identification details into a FIFO queue at the server site before returning the first RPC. Then the client process generates a signal to stop itself at this satge and waits for a wakeup signal from the server.

Stage (c)

The un-terminated child process of the system co-ordinator reads identification information from the FIFO queue and then re-activates the corresponding client process. In the meantime, the child process will suspend its execution allowing the client process to have control over the entire DCSUNIX system.

Stage (d)

The re-started client process calls the second RPC routine, access_manager of the sytem co-ordinator, in order to access an object stored previously in the PS-algol database system via the PS-algol program **service** with the data information required to carry out the operation as arguments.

Stage (e)

The access_manager receives information from the client process and then transfers them to the two Unix files cmdfile and datafile accordingly. At this point, the program **service** is invoked to perform the service requested by the client process with the data stored in cmdfile and datafile as inputs. Having completed the required service successfully, the process number of the system co-ordinator's child process and the result of the second PC are returned to the client process to terminate the whole transaction.

Stage (f)

After the client process gets the result of the second RPC back from the server, it transmits a signal, using the returned process number of the child process, to re-start the child process so that it can continue to serve another client process.

6.4. Conclusions

This chapter has presented the design and implementation issues of the distributed computing system DCSUNIX. It is a multi-user system in the sense that several client processes are able to make a request for a particular service from the system, but not actually access those abstract objects stored in the server's PS-algol database system simultaneously. Currently, DCSUNIX is only designed to cope with one object at a time in either the read or write mode. The ordering of using DCSUNIX is regulated by a First_In_First_Out queue implemented by two Unix pipes at the server site. Another interesting characteristic of this system is that it permits different types of abstract objects to be placed in the same database via the **first class procedures** mechanism provided by PS-algol.

It may also be realized that the present DCSUNIX system is not really a fully distributed system because the client processes have to specify the location of the server machine in order to make the two remote procedure calls (queue_manager and access_manager), and therefore lacks of location transparency. Nevertheless, this is only a minor problem since the main theme of this thesis is to investigate the tradeoffs between structured and un-structuerd remote data access methods. Besides, it is primarily the task of a name server.

Furthermore, there are also some controversy over the number of remote procedure calls (RPCs) used during the establishment of DCSUNIX. A corollary is that the fewer the number of RPCs, the better the performance of the system as it reduces the communication overheads. However, the main reasons of rejecting the single RPC approach are now discussed.

Although the single RPC strategy does not require any FIFO queue for access control because a client process can start updating the required abstract object as long as the server is idle, there is a high probability that a particular client process which demanding a hugh amount of data from the database system will block the other client processes from using the system for a considerable period of time. Apart from degrading the throughput, this also makes DCSUNIX virtually become a single-user system. In addition, it is very likely that each client processes has to spend a lot of processing time just to test when the system is available before actually performing any access operations.

Unfortunately, the two RPCs approach also bears a big drawback concerning with orphan processes. If the DCSUNIX system crashes (may be due to power cut) somewhere between the invokations of the two RPCs, then all the identification information kept in the FIFO queue will be lost. Consequently, each un-served client processes become an orphan. The only remedial action to be taken is to re-start all the processes involved and repeats the whole transaction. However, the problems of orphans is outside the scope of this thesis and hopefully it should happen very rarely.

Finally, DCSUNIX is not a fault-tolerance system in the sense that it makes no effort to tackle any error found during a transaction. It will just return an error message to indicate the occurrence of the fault and then terminates.

Chapter Seven -- Testing of the implemented distributed system

Having presented the design and implementation details of the distributed system DCSUNIX in the last chapter, this chapter will describe a series of experiments that are carried out using DCSUNIX as the testbed. All these experiments can be classified into two categories according to the application test programs: the stack and the students data bank applications. Two extreme access strategies, the structured and un-structured remote data access methods, are applied to each of the two applications during the course of the testing.

The main objectives of the testing are:

(a) to determine the accuracy and efficiency of the DCSUNIX system;

(b) to assess the advantages and disadvantages of the two access methods mentioned above;

(c) to gain an in-depth view of the technique required for performing networking experiments.

This chapter consists of three parts. The first part outlines the scheme undertaken in DCSUNIX for the testing. The second part addresses the implementation issues of the two applications of DCSUNIX, followed by a discussion of the results obtained in the final part.

7.1. The testing scheme

First of all, in order to demonstrate the ability of DCSUNIX, a variety of experiments are tested against some of the well-known operations of the two chosen examples mentioned above, such as push or pop a data item into or off the stack, retrieve or update a specific record of the students data bank etc, using both the structured and un-structured access methods in turn.

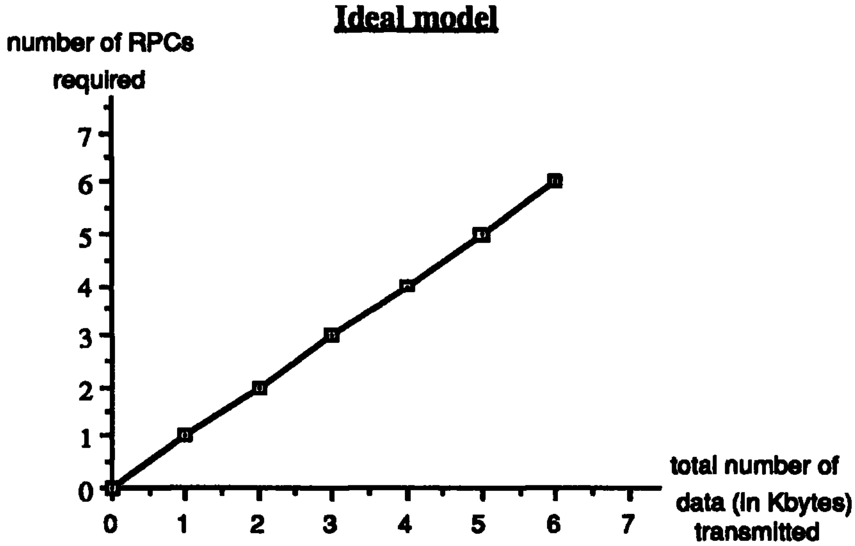
After establishing confidence with the functional behaviour of the DCSUNIX system, the two access methods are used again with the two examples but they will be operated on a unique standard request this time. That is, some information (composed of the command required to carry out the request and the data to be transmitted) will be sent from a client process to DCSUNIX's server which will, in turn, update the appropriate abstract object in the PS-algol system as desired.

Initially, for each of the two chosen examples and for each of the two access methods, a null request is sent which contains only a null command without any effective data. The purpose of doing this is to measure the communication overheads in both cases. Then this process is repeated 10 times with an increment of 1K bytes of information including both the command and the data each time. The reason of using an interval of 1K bytes is because this is the maximum capacity supported by the current RPC communication protocol (SUN's User Datagram Protocol) for a single transmission.

The prime objective of these 11 experiments is to determine the CPU times elapsed for their transmissions. To improve the accuracy of the results, each of them is repeated 100 times and then an average time is evaluated as the final answer. Consequently, four sets of readings are obtained. Note that all the experiments are carried out under light network loading conditions (about 9 p.m in the evening) so that the maximum efficiency of the network is provided. Furthermore, all the timings are started from the point where the client process has received a restart signal from the child process of the system co-ordinator (see section 6.3.2.2. of chapter six). Note also that all the number of bytes quoted above do not include the basic components of an ordinary packet such as the packet's header, checksum, packet's trailer, etc., because they will be enforced by the communication protocol automatically on every packet.

Prior to the analysis of the four sets of experimental results, an ideal model is established for each so as to provide a guidance of the analysis. This can be achieved using the relationship between the number of RPCs needed and the number of bytes transmitted as the following graph indicated:

Ideal model



Hence, the ideal shape expected from each set of the experiments is a straight line that passed through the origin. Obviously, the latter is impossible due to communication overheads imposed by the network and therefore only a straight line is expected.

Having established the ideal model, each set of the experimental results is analysed as follows:

(a) Each set of results are plotted with the number of bytes transmitted as the x-axis and the average CPU time elapsed as the y-axis.

(b) A best-fit straight line is determined for each set of experiments using a sophisticated graphic package known as Cricket Graph produced by the Cricket Software Incorporation running on the Apple's Macintosh machines.

(c) Justify the behaviour of each set of the experiments.

7.2. Applications of the DCSUNIX system

Before performing any experiments with DCSUNIX, the following issues are worth considering:

(a) the machine type of DCSUNIX's server,

(b) the directories where the C module **system co-ordinator** and the PS-algol program **service** will be placed,

(c) the location of the two Unix files: **cmdfile** and **datafile**.

According to the network environment of DCSUNIX as shown in Figure 6.2 of chapter six, only two types of machines are provided: workstations and hosts. Theoretically, either of these machines may be designated as the server. However, for the reasons of hiding directory structure and providing a forward mechanism when a subtree in a *namespace* is moved, the Unix 4.2 BSD operating system (the current operating system of DCSUNIX) often uses a special naming feature known as **symbolic link** to identify objects such as files, directories, etc., and therefore there is a high probability that the directory where the PS-algol database system mounted is referenced by such a link (actually this is the case when DCSUNIX was developing). Unfortunately, symbolic links possesses some unpleasant problems:

(a) The semantic of ".." (the parent of a context) in the presence of a symbolic link. Suppose "/user" is a symbolic link to "/usr/wct", does "/user/.." denote "/" or "/usr" ? The answer will solely depend on whether ".." is interpreted statically or dynamically. Since ".." is simply a special entry in a directory and Unix keeps no record of the path by which a given context was reached, its interpretation would work more naturally with symbolic links. Consequently, "/user/.." is interpreted as "/usr" rather than "/". This can cause unexpected anomalies with pathnames of the form "../x" when the context has been reached unknowingly via a symbolic link.

(b) A symbolic link has another curious characteristic. As its value is just a pathname and if this begins with a "/", it is interpreted relative to the root as might be expected. However, if on the contrary, the pathname contained in the symbolic link does not start with a "/", it will be interpreted relative to the directory in which the link is found rather than the current directory. Thus, absolute symbolic links are in fact relative to a dynamic definition of the root which may have changed since the link is created, whereas relative symbolic links are actually absolute because they are not affected by the definition of the root or the current directory at the time when the kink is resolved. This distinction is of particular important in a distributed system where processes from different sub-systems may have different definitions of the root and may therefore interpret the same symbolic link in a different ways.

Nevertheless, the problems of symbolic links can be overcome by employing a host as the server (bylands currently) because symbolic links are always interpreted relative to the root by this type of machine. Therefore, the PS-algol database system is always accessible.

The last two issues mentioned at the beginning of this section are relatively less restrictive. The two modules **system co-ordinator** and **service** can be situated anywhere within the user directory area provided that access to the PS-algol database system has been acquired. On the other hand, the best place for the two Unix files **cmdfile** and **datafile** would be in the same directory as the PS-algol database system on the ground of compactness.

So again, it can be realized that the PS-algol database system is such a vital element to the entire DCSUNIX system. The following two sub-sections will describe two practical applications of DCSUNIX which are the stack and the students data bank examples.

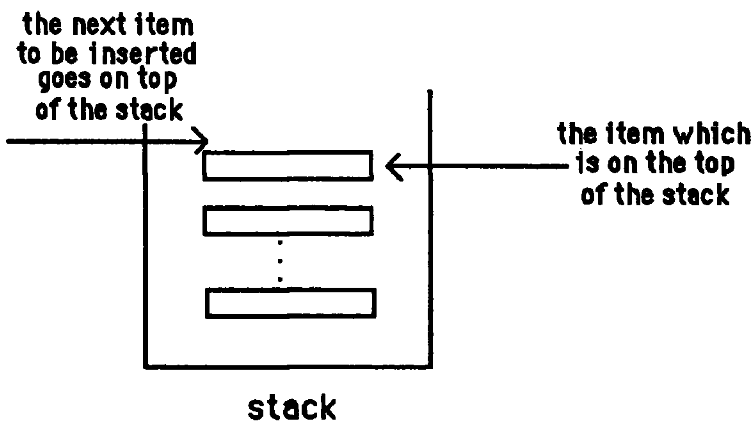
7.2.1. The stack application

7.2.1.1. Specification

The first application of the DCSUNIX system is an object (abstract data type) commonly known as a **stack**. The main reason of choosing this kind of object is due to its simplicity and commonness. A brief description of such an object is as follows:

A stack is a collection of data kept in sequence. Each item of data is of the same type. Data is added to and remove from the sequence at one specific end of the sequence (usually called the **top**). It is possible to access only the top item of data in a stack.

Pictorially, a stack can be viewed as:



Four stack operations, the only means that the stack is accessed, are considered. They are:

(a) push - an operation that, given an item, inserts that item at the top of the stack;

(b) pop - an operation that deletes the top item of the stack if it is not empty;

(c) top - an operation that returns the item at the top of a non-empty stack as its result;

(d) empty - an operation which returns the string "stack is empty" when the stack is empty, otherwise the string "ok" is returned. This operation usually used prior to the operations pop and top described above by DCSUNIX internally for self-checking purposes.

7.2.1.2. Implementation

In order to test the DCSUNIX system, a stack called "StringStack" which contains a collection of data items kept in sequence is implemented. Each data items contains a string of maximum 20 characters and a pointer which pointed to the next item of the stack. The implementation process is split into two phases. In the first phase, the internal representation of the stack (the string and the pointer) together with the implementations of its four associated operations (those described in section 7.2.1.1.) are established and then stored in a PS-algol database called "utility" with entry name "application1" via the PS-algol module **utility.class.lib.**

The second phase of the implementation process requires the co-operation of the PS-algol program **service** which has been described in chapter six. The definition of the stack, i.e. the headings of its four operations, are included in the declaration part of **service** in order to allow the stack to be available to the outside world.

Note that two versions of each of the two PS-algol modules mentioned in this sub-section exist and their code can be found in Appendix D. Also, these two modules are shared between this stack application and the next application of DCSUNIX which will be described in section 7.2.2. so as to form a library of applications for DCSUNIX.

7.2.1.3. Testing

According to the test strategy given in section 7.1, the implemented stack is tested in two separate stages. For the first stage, the following requests are carried out with the stack using both the structured and un-structured remote data access methods in turn:

(a) attempt to top an item off the stack when it is empty,

(b) a client process is employed to push the same string into the stack for two successive times,

(c) repeat the task of (b) above but this time the same client process is invoked simultaneously from two different machines,

(d) and finally, all the inserted strings are retrieved from the stack to determine the integrity of DCSUNIX.

In the second stage of the testing as suggested in section 7.1, the stack is used again repeatedly with the two access methods to send information from a client process to the server. Effectively, the only difference between this set of experiments and experiment (b) mentioned in stage one is the dimension of the variable where the outgoing information is stored. Test programs (C modules) for some of the experiments mentioned in this sub-section can be found in Appendix D.

Finally, it should be emphasized that from the software re-usability point of view, the structured remote data access approach always imposes more constraints than the un-structured approach in performing the above four requests. This is primarily due to the that it is very hard to define uniquely the two modules of the server, **system co-ordinator** and **service**, such that the stack could be accessed independently without being aware of the nature (type) of the data items placed on the stack, because the two modules have to know the nature of the data items (not the internal representation of the stack) in order to construct the appropriate data transfer routines. Similarly, when a client process wishes to access the stack, it also has to supply the corresponding types of parameters to the two modules. As a result, whenever a new stack is encountered, the two modules must be re-tailored to meet the new requirements.

On the contrary, the un-structured approach enables the stack to be accessed independently since it only entails to know the starting memory address of a variable where the data is situated, and an integer which indicates the number of bytes required from that address as illustrated by the test programs in Appendix D. Because of this independency, it is possible to construct a single C module the **system co-ordinator** and a single PS-algol program **service** that will be suitable for accessing all kinds of stacks (actually this concept can be applied to all kinds of abstract objects). Unfortunately, one weakness of this approach is the loss of certain degree of data security.

7.2.2. The students data bank application

7.2.2.1. Specification

To enhance one's confidence with the DCSUNIX system, another application of this system is presented.

In this second application, another abstract object named "students.data.bank" is used which aims to maintain a data bank for students studying at a university during academic year 1988/1989. A unique personal record has been allocated to each students and the following operations are the only means that the data bank can be accessed:

(a) insert - an operation that, given a new student's record, adds that record to the data bank,

(b) delete - an operation which, given some kind of identification details of a student record, removes that record (if it exists) from the data bank,

(c) search - an operation that takes some kind of identification details of a student record and then retrieves all the information stored inside that record if it exists,

(d) update - an operation that, given the identification details of a student record and the new information of that record, replaces that student's old record in the data bank with the new one.

(e) single.write - an operation that, given the identification details of a student record and the new information of a particular field of that record, overwrites the previous information stored in that particular field by the new information supplied.

(f) single.read - an operation that, given the identification details of a student record and the field required, reads the information stored in that particular field of that record.

7.2.2.2. Implementation

Each student is represented by a record in the data bank which consists of the following information: the name, age, sex and address of the student, and the details of the major course attending by the student. All these components are stored in three parts.

The first part gives the overall structure of the record called **student.record** which contains the following fields (and they are written in a combination of bold and italic text style):

(a) ***name*** - a string of maximum 30 characters;

(b) ***age*** - an integer;

(c) ***sex*** - a single character : F or M;

(d) ***address*** - a pointer that is used to reference the second part of the record as explained later;

(e) ***course*** - another pointer which is used to point to the third part of the record;

(f) ***other*** - a pointer reserved for future use.

The second part defines the structure of the ***address*** field of structure **student.record**, named as **student.address**. The following fields can be found in this structure:

(a) ***house.no*** - an integer which records the house number of the student's home address;

(b) ***street*** - a string of maximum 30 characters which stores the street name of the student's home address;

(c) ***town*** - a string of maximum 20 characters that keeps the town name of the student's home address;

(d) ***next.address*** - a pointer which is reserved for future modifications.

Finally, the third part of a record defines the structure of the ***course*** field of structure **student.record**, called **student.course** which contains:

(a) ***department*** - a string of maximum 30 characters to keep the name of the department that the student is registered;

(b) ***year*** - an integer to store the student's current year of study;

(c) ***next.course*** - a reserved pointer which may be used to store details of another course taken by the student.

All the structures described above are implemented by means of PS-algol's structure type and the field ***name*** of structure **student.record** is designated as the only means by which a particular student record can be identified. There are two key advantages of dividing the definition of a student record into several parts. First, it is easier to construct a student record. Second, it allows extra information to be added subsequently via the additional pointer field(s) in each parts of the record because a PS-algol pointer can be used to point to any types of PS-algol structures (see reference [b] of Appendix B).

As in the stack application, the implementation process of this data bank application is performed in two phases using the two PS-algol modules **utility.class.lib** and **service** again except that the entry name of the abstract object **students.data.bank** in the database "utility" is "application2".

7.2.2.3. Testing

There are also two stages of testing for the data bank application. At stage one, the following requests are carried out with the abstract object, students.data.bank, using both the structured and un-structured remote data access methods in turn:

(a) attempt to retrieve a student record from the data bank which is currently empty;

(b) insert a list of three different students' records into the data bank by means of three separate client processes;

(c) search each of the three inserted records to assure their existence in the data bank;

(d) update the contents of two of the inserted records;

(e) retrieve data out of the individual fields of a particular student's record one by one;

(f) attempt to modify a particular field of a student's record;

(g) choose a particular student record as the target and then performs the following operations:

(1) a client process is called from a machine which attempts to read the target record while another client process is invoked from another machine trying to update the target record at the same time;

(2) repeat operation (1) several times to observe the effect.

(h) and finally, remove all the student's records from the data bank, followed by another search operation to assess the integrity of DCSUNIX.

N.B. (1) Experiments (e) and (f) above will only be tested when the structured access approach is applied.

(2) Since there is no structure imposed on a student record when the un-structured access method is applied, an extra item is therefore needed for identification purposes.

In the second stage of the testing, as in the previous stack application, information is sent (i.e. only operation insert is involved) from a client process to the server at an increasing rate of 1K bytes for 11 times, except the first one is a null request, using the two access methods in turn. Again, each experiments is repeated 100 times to obtain an average value. However, in order to accomplish this two sets of experiments, the overall dimension of a data bank record must be adjusted. This is achieved by changing the size of some of the individual fields of the record, such as ***name***, ***street***, ***town*** and so on, but not the general structure of the record.

Finally, some of the test programs (C modules) for this data bank application can be found in Appendix D.

7.3. Discussion of the results

Since the two applications of the DCSUNIX system, the stack and the data bank examples, have been tested in two separate stages, and therefore this section will be divided into two parts. Each of them discusses the results obtained from the two examples.

7.3.1. The first stage

All the expected results of the experiments mentioned in section 7.2.1.3. and section 7.2.2.3. have been obtained apart from experiment (g) of section 7.2.2.3. as indicated in Appendix E. The main reason is that the result of this particular experiment is affected by the arrival time of the two client processes' requests which will, in turn, determine their positions in the server's FIFO queue. Additionally, the arrival time itself can also be affected by the availability and efficiency of the network. Therefore , it is very difficult to predict the outcome. However, the results of this experiment has demonstrated a very important characteristic of the DCSUNIX system. That is, the FIFO queue of DCSUNIX has provided an efficient way of maintaining consistency of the states of the abstract objects stored in the PS-algol database system, yet permitting multi-user processing. The ultimate effect is to put a lock on every transaction (read or write) and this lock will not be released until the transaction has committed. Unfortunately, this approach has sacrificed the traditional "single writer or multiple readers" type of locking mechanism.

During the testing process, two minor problems are experienced from the user point of view:

(a) Whenever a string is to be transmitted from a client process, a double quotes symbol " must be included in both ends of the string in order to validate the syntax of a string in the language PS-algol.

(b) The statement "pr_open: FBIOGTYPE ioctl failed for /dev/fb" always appears on the screen of the server's machine during the execution of the PS-algol program **service** which has been clarified as a local network setup defaults and can be ignored.

Up to this stage, the implemented DCSUNIX system has given encouraging results.

7.3.2. The second stage

This is the most interesting part of the whole testing procedure because it shows some indication about the magnitude of the performance of DCSUNIX subject to the two access methods. All the numerical results obtained from the two applications of DCSUNIX are shown by Table 1-4 in Appendix F.

Using the data from these tables, four graphs are plotted, as depicted by Figure 1-4 in Appendix F respectively, with the total number of bytes of information transmitted as the x-axis and the average CPU time elapsed as the y-axis. Notice that a best-fit straight line has been included in each graphs.

From the offsets between each set of the experimental results and their corresponding best-fit straight line, it is evident that they exhibit a similar behaviour to that predicted by the ideal model established in section 7.1. In addition, it can also be observed from the graphs that the offsets tend to deviate from linearity as the total number of bytes transmitted is approaching 8K bytes. A logical explanation for this type of behaviour is because of communication overheads which are a combination of access and *throughput* delays. When the number of bytes transmitted is low, the number of RPCs required is also low and therefore the communication overheads are small. However, as the number of bytes transmitted increases, the total number of required RPCs will grow and at the same time the communication overheads starting to accumulate. This kind of delay would still be tolerable up to a point where the total delay time is so huge and becomes noticeable. For the DCSUNIX system, this occurs at approximately 8K bytes and this point may then be described as the "fading point" of the system.

Besides, the best-fit straight lines determined from each graphs can also be used as a tool to assess the performance ratio between the structured and un-structured remote data access methods by comparing the gradients of the respective lines. In case of the stack example, the gradients of the structured and un-structured methods are 0.0876 and 0.0433 respectively; whereas in case of the students data bank example, they are 0.0761 and 0.0404 respectively. Hence, the estimated performance ratio of the stack example is approximately 2.02 while the performance ratio of the data bank example is about 1.88. It can be realized that the un-structured approach is always faster than the structured approach. This gained efficiency in the un-structured approach is mainly contributed by the absence of record and field boundaries.

Conclusively, the most important characteristics from this thesis point of view is the measurable performance of the implemented distributed system DCSUNIX. As illustrated by the results obtained, it is reasonable to allow DCSUNIX to be employed with confidence as a test-tool for the investigation of the structured and un-structured remote data access methods in distributed computing systems.

Chapter Eight -- Conclusions

Distributed computing systems consisting of single-user machines, eg. workstations, connected by a fast local-area network are becoming very popular. Many of them have been employed for creating general-purpose computing and information processing environments such as CFS [35], XDFS [38], WFS [78], AFS [79], ALPINE [80] and VICE [81]. For economical reasons, the machines need to share resources. In particular, it is necessary to facilitate sharing of files and databases. Therefore, the management of shared resources is an important service that should be provided by a trusted authority. Since the workstsations are controlled by the their users, they cannot be guaranteed to be always available or be fully trusted. An obvious solution is to designate some of the machines as the servers to administer the shared resources and support applications (services) running on the system. For simplicity, only systems possessing such a single server machine are of interest.

One of the major design goals of any distributed systems is to provide users some kind of remote access to objects stored at different sites of the network. The term **object** is perceived as a collection of data of the same type which can only be accessed through a well-defined interface. Because of this invisibility, they are referred as the **abstract objects**. Abstract objects are implemented in terms of abstract data types and they are often needed to persist over a period of time until they are no longer required. Two modes of access to these abstract objects are considered: the **structured** and **un-structured** remote data access models. In case of the former model, data is simply treated as rows of bytes whereas in the latter model, data can only be accessed via an appropriate access procedure. This thesis is designated as a comparative study of these two access models.

8.1. Achievements

As a preparatory stage, the fundamental concepts behind distributed computing systems, abstract data types and persistent data type systems are introduced. Subsequently, a small-scale experimental multi-user distributed system known as the DCSUNIX system is established as the testbed for the comparison of the two access methods. Finally, some measurements are taken using the implemented system via two specific applications: the stack and the students data bank examples.

The testing strategy is described briefly as follows. First of all, a variety of experiments are tested against some of the well-known operations of the two chosen applications accordingly, such as push or pop an item into or off the stack, retrieve or update a particular record of the students' data bank etc, using both the structured and un-structured approaches in turn so as to show the accuracy and reliability of the implemented system. Then, the two access models are used again with the two examples but this time, they will be operated on the same standard request which involves the transmission of some information (composed of the command required to carry out the request and the data to be transmitted) from a client process to the server of DCSUNIX.

Initially, a null request (which contains just a null command without any effective data) is sent to measure the communication overheads. This process is repeated 10 times with an increment of 1K bytes of information including both the command and data each time. As a result, four sets of readings are obtained by evaluating the average CPU time elapsed including the communication overheads for each experiment and then four corresponding graphs are plotted with the number of bytes transmitted as the x-axis and the average CPU time elapsed as the y-axis. Finally, the tradeoffs of the two access methods are determined through the comparison of the four graphs obtained to their respective theoretical estimates established before the testings. Note that all the testings are carried out under light loading conditions of the network and all the timings would not start until the client process has received a re-start signal from the child process of the system co-ordinator at the server site. Note also that all the number of bytes quoted above do not include the basic components of an ordinary packet such as the packet's header, checksum, packet's trailer, etc., because they will be enforced by the communication protocol automatically on every packet.

Consequently, the following points are concluded:

(a) As illustrated by the four graphs in chapter seven, the DCSUNIX system has represented a quite satisfactory model for the analysis of structured and un-structured remote data access strategies.

(b) From careful study of the underlying principles of the two access methods in conjunction with the results obtained from the experiments, the final verdicts of this thesis can be summarized as follows:

(1) For the stack example, the average access time of the un-structured approach is found to be about 2 times faster than the structured approach; whereas for the students data bank case, the average access time of the un-structured approach is 1.8 times faster than the structured approach. This is primarily due to the absence of record and field boundaries in the un-structured models. Therefore, it may be concluded from these results that the un-structured remote data access method is approximately 2 times faster than the structured remote data access method subject to the current testing conditions of DCSUNIX.

(2) Since there is no structure imposed, it is very unlikely to perform any type-checking operations with the un-structured model which is in contrast to the structured case.

(3) It is extremely difficult and risky for the server to access the individual fields of a record via the un-structured method because all the information is just stored as rows of bytes in the database; however, this could allow different types of abstract objects to be stored in the same database. On the contrary, the structured method would update each field of the records efficiently.

(4) From the software re-usability point of view, the un-structured model is superior to the structured one since only one single set of generalized modules (the C module **system co-ordinator** and the PS-algol program **service etc**) is required at the server site which can cope with all kinds of abstract data types and therefore it is more flexible. But from the data integrity and security viewpoints, the structured approach is better because the system is aware of the types of data expected.

(5) When an instance of any abstract object is being stored or retrieved using the un-structured strategy, it is the user's responsibility to supply the appropriate starting address and the correct amount of bytes required. On the other hand, this responsibility is devoted to the server in the structured model as it already knows all the internal representation of the objects.

(c) Finally, it should be noted that the existing DCSUNIX is not a fault-tolerance system. Once an error is detected, perhaps due to unsuccessful RPCs, the whole transaction must be aborted to preserve the effect of atomic transactions (see section 2.3.4.2.).

8.2. Further improvements

In the current implementation of the DCSUNIX system, several users are capable of making a request for service from the system simultaneously but thereafter they have to wait until the server is idle. Under light network conditions, the suspension time is short enough to give a satisfactory response. Nevertheless, three possible features can still be added to the present system.

8.2.1. First improvement

There is no special reason why DCSUNIX cannot be extended to deal with more than one abstract object during a single transaction as the same principle applies with the same degree of consistency since there is still only one user using the system each time. To achieve this goal, the following modifications are demanded:

(a) The two Unix files cmdfile and datafile will require more storage space depending on the total number of abstract objects involved.

(b) For the structured access model, the second RPC routine, access_manager of the system co-ordinator at the server site, must be re-constructed which would be quite complicated. Again, there is no way to generalize this module. Apart from knowing the total number of abstract objects involved, it also needs to have knowledge about the internal representation of all these objects in order to retrieve or update the objects. On the contrary, the reconstruction process of this RPC routine will be simple for the un-structured approach with only the expense of prolonged transferring time.

(c) The PS-algol program service also demands some modifications. An extra outer loop must be present so as to obtain information about all the operations that have to be performed on the corresponding abstract objects.

8.2.2. Second improvement

In the light of many distributed systems having mechanisms to cope with **orphans**, it is always a challenge to make DCSUNIX become one of them. Orphans are unwanted executions that can often manifest themselves due to communication or node failures. The former can be solved using the time-driven mechanism presented by Mckendry [82]. However, his method was based on clocks local to each site of the network and it performs best when clocks are synchronized, although non-synchronized clocks do not produce inconsistencies. Hence, it is not suitable for orphans produced due to node failures. By storing a modified First_In_First_Out (FIFO) identification queue on stable storage (perhaps in the PS-algol database) whenever it is accessed (read or write), DCSUNIX could lend itself an efficient way of treating this kind of orphans. Instead of just keeping information about **un-served** client processes, those that have been registered on the queue but have not started its transaction, a new version of the queue is established by including identification details of the client process which is in operation (if exists) before a crash. Consequently, orphans are tackled using this queue together with the co-operation of the client machines. Before proceeding any further, one assumption must also be made. That is, no client process could ever become an orphan before it has been registered on the server's queue via the first RPC routine queue_manager, otherwise it can only be removed by a suitable garbage collector mechanism.

Eventually, there are three possible situations where an orphan can be produced during a communication session: the server's machine crashed, a client's machine crashed or both of them crashed.

In the face of the first situation, since the server always puts the FIFO queue on stable storage which contains all the client processes's identification details including the one (if exists) that is in operation just before the crash, this would eliminate orphans completely provided that the second RPC's total timeout interval has not expired. The explanation is as follows. After the crash, the server will reboot itself and resumes its normal operations with the queue. To the un-served client processes, the crash will be just regarded as a long delay. However, if a client process has already started its transaction during the crash, the following special treatment is required. Having rebooted the server machine, the queue is checked to find out whether or not the process which is being served during the crash is still active at its local machine. If so, it must be killed and then may be restart again. Unfortunately, this may not be the ideal solution especially when the transaction will cause serious consequences such as withdrawing a million pounds from a bank.

On the other hand, it is much easier to deal with the last two circumstances mentioned above. In case of a client machine has crashed, three alternatives exist. First, if the crashed machine is belonging to an un-served client process and it is not its turn to use the system then no action will be taken. Second, if a client 's machine crashed just before its transaction, then the server will need to wait until it is rebooted before proceeding any further. Third, when a client process has begun but not yet completed its transaction during the crash, then the server has to kill this process using the information of the queue after its machine is rebooted.

For the last case, when both the server and client machines crashed during a transaction, the only remedial action is to reboot them before killing the client process as described previously.

All the above algorithms for destroying orphans are still valid in case of multi-machine failures. The most crucial issue of the whole strategy is the reboot time of the machines. If the reboot time is too long, it will exceed the timeout interval of the second RPC and thus harder to get rid of all the orphans produced. Note also that after a machine is rebooted, all the previously stopped (not terminated) processes are capable of resuming from their stopping points.

8.2.3. Third improvement

Another possible future work of this project will be in the direction of improving the experimental strategy of the DCSUNIX system. At the moment, experiments are only carried out at an interval of approximately 1K bytes of data which is the maximum buffer size provided by the current RPC communication protocol, SUN's User Datagram Protocol, for each transaction. So, in order to determine the best throughput curves for the two access models, more results are required which can be obtained by performing more testings on DCSUNIX with the number of bytes of data between each intervals. To eliminate overheads due to network traffic, it may be better to repeat each test, say 200 times, and then calculate the average CPU time elapsed in each case. Also, it would be interesting to elaborate the same experiments under medium and heavy network conditions so that a comparison between all kinds of network conditions can be drawn. Finally, since DCSUNIX is intended to support mulit-user processing, but at the moment only one client process (or termed as a **station**) is used to experiment the system for simplicity reason, and therefore it may be worth trying to employ more stations to re-assess the performance of DCSUNIX with the same testing scheme.

8.3. Future trends of distributed computing systems

The Distributed Computing Systems Research Programme, sponsored by the U.K. Science and Engineering Research Council and lasted for eight years from 1977 to 1984, has greatly promoted research in distributed computing. Since then, the field has been growing rapidly both in the breadth of activity and the depth of understanding. Most of the distributed computing systems in use nowadays are multi-computer configurations that do not share memory and can be dispersed over wide geographical areas. They are referred as **loosely-coupled distributed systems.**

This trend should prevail well into the future with the encouragement of current trends in hardware technologies. Among them, the semi-conductor (or chip) technology has advanced dramatically over the past few decades. Lately, the total number of transistors that can be implanted on a single chip is about 10 times as many as in 1965. The prices of microprocessors have also fallen steadily. This is reflected by the way microcomputers have evolved from the 8-bit based microprocessors such as Apple IIe and Commodore Pet 8032 which predemoinated in the 1970's, to the 16-bit microcomputers on machines like the IBM PC (Intel 8088) and Apple Mackintosh (Motorola 68000) that have prevalent the market since early 1980's. By the end of 1980's, it is very likely that the 32-bit based microcomputers will take over the dominant position of the existing 16-bit ones.

Another main need of computing is the provision of a reliable, fast and compact means of storage medium. In the past, magnetic tapes were the most practical mass storage medium and now magnetic disks dominates. However, the future of this category will be the use of optical disks [83]. Optical storage was introduced in 1978 as consumer video system based on a standard called LaserVision. The video images are stored as FM signals on the disc. Subsequently, this technology was used to produce optical audio disc known as Compact Disc (CD) on which audio information is encoded digitally. The feat of CD supported the introduction of Compact Disc Read Only Memory (CD-ROM) in early 1985. Later, Write Once Read Many (WORM) optical disks came into existence, allowing data to be written only once but could be read over again and again. The key advantages of optical disc technology over the magnetic disc technology are: much higher capacity of information can reside on similar size disc, mass replication of optical disks can be done inexpensively, optical disks are removable unlike the hard disc and they are also immune to accidental erasure and external magnetic field. Nevertheless, two shortcomings of the current optical disc technology exist: the media can be written only once and the access times of optical disc drives are slower than high-performance magnetic disc drives. Further research on erasable optical disks is in progress and hopefully an ideal mass storage medium will appear very soon.

So clearly, it is extremely tempting to connect a number of relatively cheap processors together each with its own optical disc storage medium, to achieve increased processing power, geographical separation and increased reliability. To accomplish this goal, a wide area or local area network is required. In the light of current trends in distributed computing systems, it indicates the need for local area networks are becoming more common as many organizations (or companies) have devolved responsibility away from the central office towards semi-independent subsidiaries. Local tasks can then be run and controlled by the people who understand them best; they are fully responsible for the consequences. Local area networks (LANs), which are intended to provide wide bandwidth over a limited distance, have developed enormously in recent years. Such networks make use of relatively cheap methods of interconnection such as co-axial cable, twisted pairs, optic fibres, etc. Since processing installed at the locations where computing power is needed, the communication costs can be reduced. Many different system architectures are also possible ranging from the use of intelligent terminals connected to a central mainframe, to placing powerful workstations on the desks of each employee. However, there is still no agreed international standard for LANs and no single technology dominates the market yet.

As time progressed, there was an increasing requirement for networking between organizations as triggered by the success of ARPANET. Unfortunately, this was difficult with so many incompatible protocols in existence. Furthermore, it is not enough merely to agree upon on a standard protocol. The set of protocols used must also have a well-defined structure so that the responsibilities of different levels of protocol can be clearly defined with no overlap of functionality.

The key advantages of standardization are summarized as follows:

(a) If a clear, unambiguous standard can be adopted universally, then all communication software written to conform to the standard would be able to interwork.

(b) Additional equipment(s) which conforms to the standard can be installed without further modification or enhancement.

(c) Standard bodies can also be established that devote all their time and efforts to research and development into standards of the future.

Currently, the IEEE (Institute of Electrical and Electronic Engineering) LAN standardization 802 project is attempting to provide a reference model and international standards for LANs. This project spans several documents as listed below:

(a) 802.1(A) - Overview and architecture.

(b) 802.1(B) - Addressing, internetworking and network management.

(c) 802.2 - Logical link control.

(d) 802.3 - CSMA/CD access method and physical layer specifications.

(e) 802.4 - Token-passing bus access method and physical layer specifications.

(f) 802.5 - Token-passing ring access method and physical layer specifications.

(g) 802.6 - Metropolitan network access method and physical layer specifications.

Nevertheless, any international standard should be subject to various addenda and should be periodically re-issued as necessary in order to resolve any inconsistencies or ambiguities, to reflect changes of emphasis and to take due account of developments in related areas. This process may seem unnecessarily cumbersome and one may suggests that it would be better if the standard body could define the standard and then insists all members to adhere to that standard. Unless a new standard is satisfied by the majority of potential users, it will either be largely ignored or will be subject to modification. Therefore, the existence of a standard is only useful if the majority of organizations undertake to adhere to its requirements and do not seek to improve it individually. The most important point is that once a standard has been agreed upon, if it possesses any significant weakness they will become only too clear as it is put to the test of implementation and usage.

Appendix A - - Physical database design issues of DDBs

Several important physical design aspects of DDB are desrcibed in this appendix.

Implementation of the relationships between entities :

This can be greatly influenced by the software chosen but usually there are two ways: by **pointer chains** or by **indexes**. In general, pointer chains are better if the most frequent use of a relationship is to search through all the relevant information of one occurrence, eg. reading all orders for a given customer. Indexes are superior if one tries to retrieve a single record or if the search criteria are complex.

Since there are several (may be different) machines involved in a distributed system, one may use indexes and the other use pointer chains. It should not be a problem if the same DBMS is used on both machines. The DBMS can take care of any translation that is required. However, problems may arise if different DBMSs are used. A practical solution is either to standardize on a reasonable compromise solution or to forge exchange of programs and data in favour of fairly strict control by the DBA.

Data duplication :

Data duplication is a method of saving disc accesses. For example, indexes carry record keys and sometimes additional data is also contained in the records to which the indexes refer. The point is to carry out searches using the indexes only without reading the records themselves. Data duplication is sometimes the result of a finely balanced judgment between retrieval speed and updating costs especially if there are extra disk accesses on a remote machine.

Locking :

The requirement for simultaneous access and update by asynchronous tasks has led many DBMS software vendors to implement **locking** at file level or even record level recently. Generally, the lower the level of locking, the lower the probability of contention and hence deadlock. In a distributed system, if a task on machine A accesses data on machine B, the smaller the amount of data locked the better since a task that accesses data remotely is likely to run for a long time. However, in practice, two further problems needed to be considered:

(a) If the remote part of a navigation path (traverse) has to stop because of contention, how does one ensure that the task originated the access does not get time-out?

(b) How does one detect deadlock if it arises out of tasks that are running on different machines?

More research work is needed to solve these problems but Gross et al [a] has suggested that a non-dynamic form of locking (eg. locking a whole file) may be the best policy to adopt.

Data placement designs :

Three types of data placement designs can be used for DDB :

(a) A **partitioned** system design: Each network node is allocated disjoint sub-sets of the organization's database and access to data can be prohibited by limiting access to the node at which the data are stored. This design is appropriate when storage space is limited at some sites.

(b) A **replicated** system design: Each network site retains a complete copy of the database and the distributed DBMS must extensively co-ordinate the synchronized updating of data. This approach is frequently used when infrequent updating is sufficient for data processing applications; updates at one node are periodically broadcasted to other nodes. This strategy has great reliability as current data can normally be accessed from any location, and has high data retrieval efficiency as accessing is localized. The only cost for this design is storage spaces.

(c) A **clustered** system design: Each node may have unique sub-sets of the database as well as selected redundant copies of some files or sub-sets of files.

Privacy :

In a distributed system, one may wish to restrict access and update rights of remote users if they are the privileged users, eg. database administration staff or system programmers. However, a distributed system could be more dangerous in that an **intruder** may be able to masquerade as another node in the system rather than simply to pose as a legitimate terminal user: by posing as another node he/she may obtain bulk data in a short period of time. Two precautions can be taken to deal with intruders :

(a) encryption of the communication lines to a sufficiently high standard;

(b) monitoring of traffic between nodes, to detect whether any response is given by one node that does not seem to have originated from another node. Some automatic checks, eg. an intra-node secret code, may be built into the protocol. If the expected code is not received, an alarm message is generated.

However, all privacy precautions are expensive in time, in hardware and in software, and therefore the expense that can be justified must be related to the value the information in the system could have for an intruder.

Reference

[a] GROSS, J.M., JACKSON, P.E., JOYCE, J., and MCGUIRE, F. A. 1980. "Distributed database design and administration". Cambridge University Press, pp. 285-296.

Appendix B -- General information about the PS-algol system on SUN Unix 4.2 Release 3.4

1. Components of the system

The system consists of the following three sets of files:

(a) In directory /usr/local/psalgol/cmd (EXECUTABLE FILES)

NAME	FUNCTION	CONTENTS
psb	PS-algol browser	psr /usr/lib/ps/browser.out
psc	PS-algol compiler	psr /usr/lib/ps/PScomp.out $* | cat
pse	PS-algol elide program	psr /usr/lib/ps/elide.out $* | cat
pso	PS-alogl dis-assembler	psr /usr/lib/ps/psops.out $* | cat
psr	PS-alogl interpreter	machine codes
sc	pseudo S-algol compiler	PSPRELUDE+/usr/lib/ps/Sprelude export PSPRELUDE psr /usr/lib/ps/Scomp.out $* | cat
sr	alias to PS-algol interpreter	psr $*
stag	PS-algol garbage collector	machine codes

(b) In directory /usr/local/psalgol/lib (PS-algol library)

Since the files in this directory contain either machine codes or a mixture of machine codes and PS-algol statements, and they are not concerned very much in executing PS-algol programs, so only their functions will be described briefly.

NAME	FUNCTION
PScomp.out	PS-algol compiler
Scomp.out	pseudo S-algol compiler
Sprelude	prelude for pseudo S-alogl compiler

browser.out	PS-algol browser program
build	shell script to construct the database directory
buildcurs.out	program to load a PNX cursor image
buildfont.out	program to load a PNX font
elide.out	PS-algol elide program
poms.out	PS-algol initialisation program
prelude	PS-algol prelude
psops.out	PS-algol dis-assembler program
stand	list of PS-algol standard identifiers for users
update	shell script to update the database directory

(c) In directory /usr/local/psalgol/lib/init (PS-algol setup programs)

NAME	FUNCTION
ALL	help text used by the browser
cou20.kst	PNX font "courier 20"
db.browse.out	loads the browser
dbtof.out	copies text from a database to a file
errors.out	loads the error records
events.out	loads the events structure
fix13.out	PNX font "fixed 13"
ftodb.out	copies text file into a database
gac16n.kst	PNX font "gac 16n"
hci45i.kst	PNX font "hci 45i"
help.out	loads the help utility
intcomp.out	loads the compiler used by the browser
more.out	loads the more utility
outline.out	loads the outline graphics
padfile	extends a file's length to a number of blocks
print.out	loads the print statement utilities
raster.kst	loads the raster graphics functions
simple.menu.out	loads the "Simple.menu" function
tab.trav.out	loads the traversal function for tables

tables.out	creates the system database
trav.out	loads the traverser utility

2. Running PS-algol programs

Before trying to execute any PS-algol program, users should first read the information sheet "The PS-algol terminal emulation for the SUN" in order to understand the I/O operation of the system. Secondly, all the files mentioned in section 1 must be made accessible to the users' home directory using the **set path** command in the .login file. Finally, enter the following additional commands in the users' .login file:

(a) set PSDIR=/usr/lib/ps/dbs
(b) /etc/rpc.statd
(c) /etc/rpc.lockd

The reason of doing (a) is to define a shell variable as the name of the database directory. In addition, there should be two processes running on each SUN workstation which are (b) and (c) above. These processes manage the locking of files via the file system control call **fcntl**. They will only work if a file on a SUN file server is being locked. If these two deamons are not present, the kernel calling them will wait for them to be restarted. However, nothing in the system will restart them so a call to fcntl can be delayed indefinitely.

Having done all the procedures above, programs can be run through the following steps:

(a) Compiles the program using the PS-algol compiler,

psc FILENAME

to produce a file called FILENAME.out

(b) Execute the FILENAME.out file by one of the following two ways:

(i) psr FILENAME.out

(ii) psr FILENAME.out<inputfile>outputfile

Appendix C -- SUN's Remote Procedure Call facilities

1. Introduction

SUN[1]'s **remote procedure call** (RPC) technique provides a clean, procedure-oriented interface to remote services. RPC is a high-level protocol built on top of low-level transport protocols and it does not depend on services provided by specific protocols, so it can be used easily with any underlying transport protocol. Currently, the **User Datagram Protocol** (UDP) is the only transport protocol supported by SUN for RPC applications.

The use of RPC allows a client to communicate with a remote server. In this process, the client first calls a procedure to send a data packet to the server. When the packet arrives, the server calls a dispatch routine to perform whatever service is requested, sends back the reply, and then the procedure call returns to the client. Figure B.1 summarizes the RPC paradigm described above.

Since network communications often involve more than one type of machine, it is necessary to provide a common way of representing a set of data types over a network which is the task of SUN's **eXternal Data Representation** (XDR) protocol. This protocol takes care of problems such as different byte ordering on different machines. It also defines the size of each data type so that machines with different structure alignment algorithms can share a common format over the network. Furthermore, the XDR **data definition language** is the tool by which the parameters and results of each RPC service procedure are specified. This language is very similar to the language C except that a few new constructs have been added.

---

[1] SUN is a trademark of SUN Microsystems Inc.

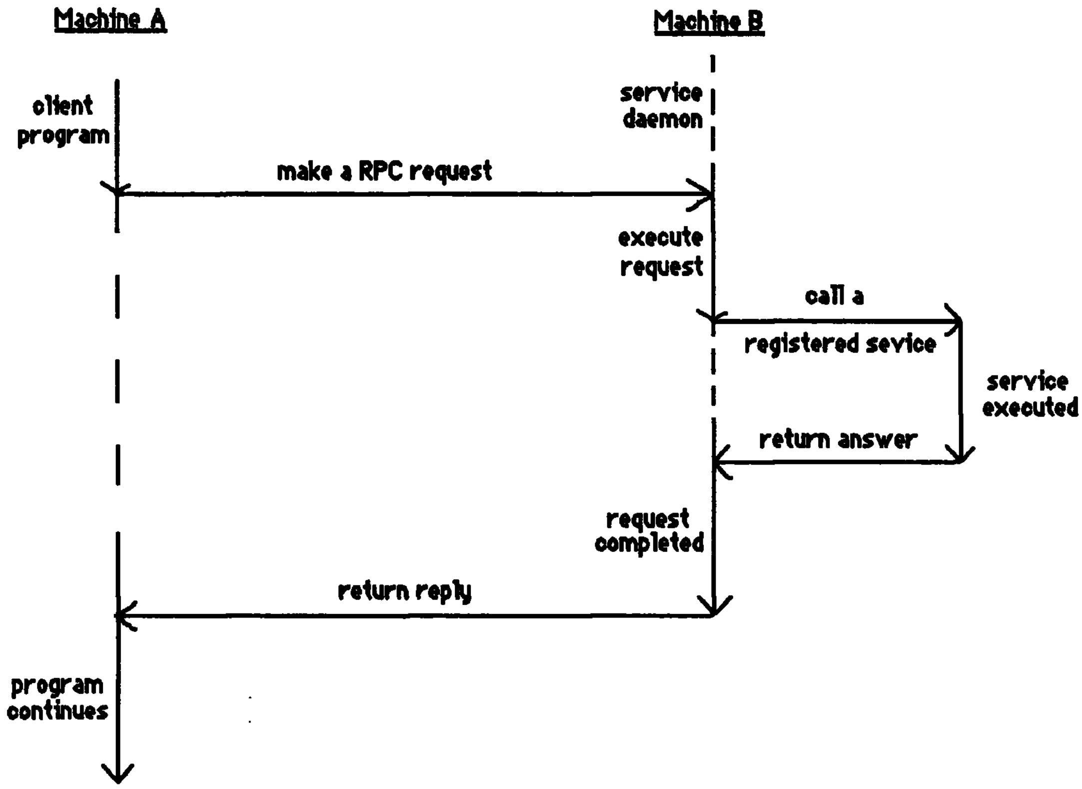


Figure B.1 Network communication with RPC

Although this appendix will only discuss the interface to C (as this is the language Unix was implemented), remote procedure calls can be made from other languages theoretically. Moreover, this appendix will focus on using RPC for communication between different processes on different machines even though it works equally well for processes on the same machine.

2. RPC layers

The RPC interface is divided into three layers as illustrated by Figure B.2. The highest layer is totally transparent to the programmers. Therefore, programmers do not have to be aware that RPC is being used, they simply make the call in a program just as other ordinary procedure calls.

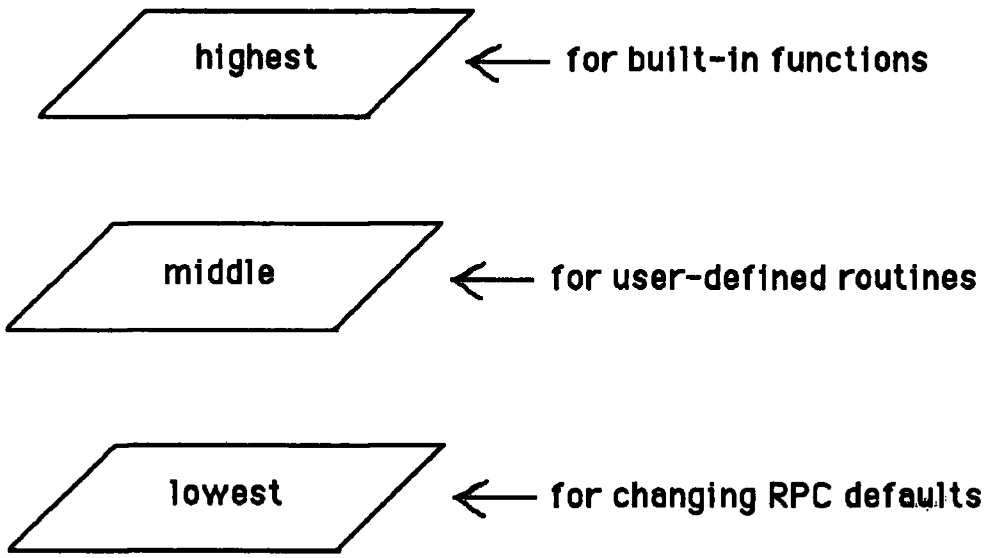


Figure B.2 The layers of RPC

At the middle layer, the routines **registerrpc** and **callrpc** are used together to make remote procedure calls as explained in section 3. These two middle layer routines are designed for most common applications and shield user from knowing about **socket** which is an end-point of communication to which a name will be bounded.

Finally, the lowest layer is aimed for more sophisticated applications such as altering the defaults of the middle layer routines. At this layer, programmers can explicitly manipulate sockets that transmit RPC messages.

3. The RPC technique

For most RPC applications, the middle layer routines, **registerrpc** and **callrpc**, will be involved. These routines are responsible for the client and server machines respectively as follows.

3.1. Server side

Normally, a server registers all the RPCs it plans to handle using **registerrpc** and then goes into an infinite loop waiting for service requests by means of the standard procedure **svc_run**. The routine **registerrpc** has six parameters. The first three parameters are the program number, version number and procedure number of the remote procedure to be registered; the next parameter is the name of the C procedure implementing it. Thus, these four parameters have identified the remote procedure uniquely. The last two parameters are the types of the input and output values of the remote procedure. If the registration is succeed, **registerrpc** returns zero, -1 otherwise. However, several points are worth mentioning in using **registerrpc**:

(a) Only the UDP transport mechanism can use this routine.

(b) The UDP transport mechanism can only deal with arguments and results of approximately 1K (to be exact is 1093 by experimental evidence) bytes in length, although it has been claimed 8K bytes by SUN [b].

(c) The program number of the remote procedure to be registered should be within the **binary** numbers, 20000000 and 3fffffff, because this is the range SUN has reserved for customer applications.

Furthermore, in order to handle arbitrary data structures, regardless of different machines' byte orders or structure layout conventions, all the RPC parameters and results are converted to the network standard XDR before sending them over the wire. The process of converting from a particular machine representation to XDR is called **serializing** and the reverse process is called **de-serializing**. XDR can support both the built-in types and user-defined ones. A complete description of XDR routines could be found in [a].

As an example, the following C program will register a procedure called test on a local machine called bolton.

```
#include <stdio.h>
#include <rpc/rpc.h>

int test(counter)
int counter;
{
counter=counter+10;
}

main()
{
      registerrpc(0x20000000, 1, 1, test, xdr_int, xdr_int);
      svc_run;
}
```

In this example, the type field parameters of **registerrpc** are both **xdr_int** which is a filter primitive that translates between C integers and their external representations. Since xdr_int is a pre-defined XDR routine, nothing is needed to be done. However, for a user-defined type, a definition of that specific type must be included in the program which is made up of one or more standard XDR routines.

3.2. Client side

The simplest routine in the RPC library used to make remote procedure calls is **callrpc** which has eight parameters. The first one is the name of the remote machine to which the call is made. The next three parameters, the program number, version number and procedure number, specify the required procedure at the remote site. The last four parameters define the parameter and result of the RPC. Since data types may be represented differently on different machines, **callrpc** needs both the type of the RPC argument, as well as a pointer to the argument itself (and similarly for the result).

After trying several times to deliver a message, if **callrpc** can get an answer, it returns zero, but non-zero otherwise. The full details of all the return codes can be found in the file <rpc/clnt.h>. The delivery mechanism used is also SUN's UDP so it is always safe to use **callrpc** in conjunction with **registerrpc**. Methods for adjusting the number of re-tries require the use of the lowest layer of the RPC library routines as discussed shortly.

Finally, if the procedure test registered at bolton is to be called from another machine, the corresponding C program may be written as:

```
#include <stdio.h>
#include  <rpc/rpc.h>

main()
{  int   number=11;
   int  result;
      callrpc("bolton",0x20000000, 1, 1,   xdr_int, &number, xdr_int, &result);
      printf("The return value of the RPC is %d\n", result);
}
```

where the integer result will contain the return value of the RPC.

3.3. Conclusions

Up to this point, the RPC technique permits programmer to send arbitrary data structures over the network with the aid of XDR. However, complexity and difficulty may incur if one wants to pass more than one item because there is only one input type parameter in the specifications of **registerrpc** and **callrpc**. The only way to solve this problem is to collect all the outgoing items into a single record structure and then defines a new routine for serializing. But this will impose extra programming efforts on programmers even when two integers are going to be sent. The same arguments apply to the results of the RPC.

4. Advanced RPC programming

In the examples given so far, RPC has taken care of many details automatically. Occasionally, it may be necessary to change the default values of the RPC protocol. First, one may need to allocate and free memory while serializing and de-serializing with XDR routines [b, pp.20-21]. Second, one may want to perform authentication on either the client or server side by supplying credentials or verifying them [b, pp. 32-35]. Finally, one may wish to have control over the RPC delivery mechanism or the socket used to transport the data. Nevertheless, all these requests have to be done at the lowest layer of RPC. For the purpose of this thesis, only the last issue will be discussed.

4.1. Modification of the RPC defaults

When routine **callrpc** is initiated at a client machine, the underlying RPC protocol will deliver the data packet, supplied by **callrpc**, to the required remote machine using its default values. To illustrate the layer of from which a programmer can adjust these defaults, consider the following C program:

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netdb.h>
#include <netinet/in.h>

#define remote_machine "bolton"

main()
{
  struct hostent *hp;                        /* structure hostent is defined in
                                                <netdb.h> */

  struct timeval pertry_timeout,total_timeout;   /* structure timeval defined
                                                    in time.h>          */
  struct sockaddr_in server_addr;      /* structure sockaddr_in is defined
                                          in <netinet/in.h>          */

  int sock= RPC_ANYSOCK;               /* constant RPC_ANYSOCK is defined in
                                          <rpc/svc.h> which has also been
                                          included in <rpc/rpc.h> */

  register CLIENT *client;

  int number, result;


    number=10;
    if ((hp=gethostbyname(remote_machine))==NULL)
    {
        printf("Can't get address for the remote machine%s\n", remote machine);
        exit(-1);
    }

    pertry_timeout.tv_sec=1;           /* timeout interval for each rpc    */
    pertry_timeout.tv_usec=0;          /* call in seconds and microseconds */
    bcopy(hp->h_addr,(struct in_addr *)&server_addr.sin_addr,
          hp->h_length);                     /* construct Internet address of */

    server_addr.sin_family=AF_INET;          /* the server by putting the name*/
    server_addr.sin_port=0;                  /* family address and port number*/
                                             /* of the host. When the port    */
                                             /* is 0, the remote portmapper */
                                             /* will be consulted to get the */
                                             /* actual port of the remote    */
                                             /* program. Constant AF_INET is */
                                             /* defined in <sys/socket.h> and */
                                             /* structure in_addr is defined in */
                                             /* <netinet/in.h>              */
```

```
    if ((client=clntudp_create(&server_addr,0x20000000,1,
         pertry_timeout,&sock))==NULL)          /* create an rpc client handle */
         {                                      /* for the remote program*/
              printf("Can't create client handle\n");
              exit(-1);
         }

    total_timeout.tv_sec=1;           /* the total timeout interval of the rpc call*/
    total_timeout.tv_usec=0;          /*  in seconds and microseconds which has  */
                                      /*  the same as the timeout of each try,   */
                                      /*  so that only one rpc call is performed*/


    if ((clnt_call(client,1,xdr_int,&number,                   /* a micro which    */
         xdr_int,&result,total_timeout))!=RPC_SUCCESS)         /* calls the remote */
    {                                                          /* procedure that */
              printf("Can't make RPC \n");                     /* associated with  */
              exit(-1);                                        /* the client handle */
    }                                                          /* created above. */
                                                               /* The constant    */
                                                               /* RPC_SUCCESS is*/
                                                               /* defined in      */
                                                               /*  <rpc/clnt.h>   */
                                                               /* which has also */
                                                               /* been included in */
                                                               /*  <rpc/rpc.h>    */
    clnt_destroy(client);
}
```

In the above example, the procedure test, which has been registered at the local machine bolton as described previously, is called using the lower version of **callrpc - clnt_call.** The parameters to clnt_call are a CLIENT pointer, the procedure number, the XDR routine for serializing the argument, a pointer to the argument, the XDR routine for de-serializing the return value, a pointer to where the return value will be placed and the time in seconds to wait for a reply.

Since **callrpc** uses UDP, the CLIENT pointer of **clnt_call** is obtained by calling **clntudp_create**. The parameters of **clntudp_create** are: the server address, the program number, the version number, a timeout value between tries and a pointer to a socket. It can be realized that the total number of tries to wait for a response is determined by dividing the **clnt_call** timeout by the **clntudp_create** timeout.

Currently, the default value is 5 but in the above program it is re-set to be 1 and therefore the client can only allow to make the RPC request once. Furthermore, the final argument of **clntudp_create** in the program is specified as ANY_SOCK which means the system will be informed to choose the most suitable socket for sending out the RPC requests all the times.

Finally, the **clnt_destroy** call de-allocates any space associated with the CLIENT handle, although it does not close the socket associated with it, which was passed as an argument to **clntudp_create**. The reason is that if there are multiple client handles using the same socket, then it is possible to close one handle without destroying the socket that the other handles are using.

Reference

[a] SUN MICROSYSTEMS INC. 1986(Feb). "XDR Protocol Specification". Networking on the SUN Workstation, SUN Microsystems Inc.

[b] SUN MICROSYSTEMS INC. 1986(Feb). "Remote Procedure Call Programming Guide". Networking on the SUN Workstation, SUN Microsystems Inc.

Appendix D

```
id_info process_ID;
char command[cmdlength];
char proc_str[maxdigits];
  strcpy(example1.application,"\"StringStack\"");
  strcpy(example1.command,"\"top\"");
  strcpy(example1.dbname,"\"utility\"");
  strcpy(example1.entry,"\"application1\"");
  strcpy(process_ID.hostname,server_machine);
  process_ID.id=getpid();
  if (callrpc_queue_manager(process_ID)==TRUE)
  {
    strcpy(command,"kill -17 ");   /* signal a sleep signal to itself*/
    itos(proc_str,process_ID.id);  /* and then waiting for service.  */
    strcat(command,proc_str);
    system(command);
    reply=(struct result *)malloc(sizeof(struct result));  /* this is the    */
    reply=callrpc_access_manager(example1);                 /* re-start point*/
    if (reply->status==TRUE)
    {
       printf("The top value of the stack is %s\n",reply->uval.value);
    }
    else
    {
       printf("%s\n",reply->uval.errormsg);
    }
  }
}


/* The main program body for the second experiment
   which push a string "hello" into the stack.
     process which has the same code as this module
   will also be called from another machine named
   "ws_lodge1" simultaneously in the third experiment.

main()
{request example2;
 struct result *reply;
 id_info process_ID;
 char command[cmdlength];
 char proc_str[maxdigits];
    strcpy(example2.application,"\"StringStack\"");
    strcpy(example2.command,"\"push\"");
    strcpy(example2.dbname,"\"utility\"");
    strcpy(example2.entry,"\"application1\"");
    strcpy(example2.data,"\"hello\"");
    strcpy(process_ID.hostname,server_machine);
    process_ID.id=getpid();
    if (callrpc_queue_manager(process_ID)==TRUE)
    {
       strcpy(command,"kill -17 ");
       itos(proc_str,process_ID.id);
       strcat(command,proc_str);
       system(command);
       reply=(struct result *)malloc(sizeof(struct result));
       reply=callrpc_access_manager(example2)
       if (reply->status==TRUE)
       {
          printf("Successfully push the string\n");
       }
       else
       {
          printf("%s\n",reply->uval.errormsg);
       }
    }
}              */


/* The main program body for the fourth experiment
   which retrieves all the inserted strings from
   the stack.  This can be achieved by repeatedly
   top and push the stack until it is empty.    */

main()
{request example1,tmp;
 struct result *reply;
 id_info process_ID;
 char command[cmdlength];
 char proc_str[maxdigits];
 char message[],emptystack[];
    printf("The contents of the stack is/are  \n");
    strcpy(emptystack,"stack is empty");
    do
    {
       strcpy(example4.application,"\"StringStack\"");
       strcpy(example4.command,"\"empty\"");
       strcpy(example4.dbname,"\"utility\"");
       strcpy(example4,entry,"\"application1\");
       strcpy(process_ID.hostname,server_machine);
       process_ID.id=getpid();
       if (callrpc_queue_manager(process_ID)==TRUE)
       {
         strcpy(command,"kill -17 ");
         itos(proc_str,process_ID.id);
         strcat(command,proc_str);
         system(command);
         reply=(struct result *)malloc(sizeof(struct result));
         reply=callrpc_access_manager(example4);
         if (reply->status==TRUE)
         {
              strcpy(message,"ok");
              if (strcmp(message,reply->uval.value)!=0)
              {
                   strcpy(message,reply->uval.value);
              }
              else
              {
                 strcpy(tmp.application,"\"StringStack\"");
                 strcpy(tmp.command,"\"top\"");
                 strcpy(tmp.dbname,"\"utility\"");
                 strcpy(tmp.entry,"\"application1\"");
```

```
                                        /* structure in_addr defined in  */
                                        /* <netinet/in.h>                */
        if ((client=clntudp_create(&server_addr,0x20000001,1, /*create a rpc*/
                        pertry_timeout,&sock))==NULL)   /*client handle*/
        {                                               /* for the remote*/
           printf("Can't create RPC handle\n");         /* program.      */
           return(FALSE);
        }
        total_timeout.tv_sec=1;   /*  the total timeout interval of the    */
        total_timeout.tv_usec=0;  /*  rpc in seconds and microseconds which */
                                  /*  has the same value as the timeout of */
                                  /*  each try and therefore only one call */
                                  /*  is issued each time.                 */
        if (clnt_call(client,1,xdr_idinfo,&process_ID,    /* this is a micro */
                 xdr_void,0,total_timeout)!=RPC_SUCCESS)/* which calls the */
        {                                               /* remote procedure*/
          printf("The remote server is either");        /* associated with */
          printf(" busy or down.  Please try again ");  /* the client      */
          printf("later.\n");                           /* handle created  */
          return(FALSE);                                /* above.          */
        }
        clnt_destroy(client);      /* deallocate the space assocated */
                                   /* with the client handle         */
        return(TRUE);
}


/* This routine invokes the access_manager of the "system co-ordinator"
   at the server site so as to access the required abstract object stored
   in the PS-algol database system.                             */

struct result *callrpc_access_manager(example)
request example;
{
   struct hostent *hp;       /* structure hostent defined in <netdb.h> */
   struct timeval pertry_timeout,total_timeout;  /* structure timeval defined
                                                 in <time.h>          */
   struct sockaddr_in server_addr;   /* structure sockaddr_in defined
                                        in <netinet/in.h>         */
   int sock= RPC_ANYSOCK;       /*  constant RPC_ANYSOCK defined in
                                    <rpc/svc.h> which is automatically
                                    included by the header file <rpc/rpc.h>.
                                    the purpose of this variable is to ask
                                    the kernel to choose the appropriate
                                    socket to establish the rpc connection.*/

   struct result *reply;

   char command[cmdlength];
   char proc_str[maxdigits];

   register CLIENT *client;

        if ((hp=gethostbyname(server_machine))==NULL)
        {
```



```
         printf("Can't get address for remote machine %s\n",server_machine);
         printf("Please try again later.\n");
         exit(0);
      }
      pertry_timeout.tv_sec=6;      /*  timeout interval for each rpc    */
      pertry_timeout.tv_usec=0;     /*  call in seconds and microseconds */
      bcopy(hp->h_addr,(struct in_addr *)&server_addr.sin_addr,
            hp->h_length);           /* construct Internet address of */
      server_addr.sin_family=AF_INET; /* the server by putting the name*/
      server_addr.sin_port=0;        /* family address and port number*/
                                     /* of the host. If port number  */
                                     /* is 0, then the remote portmapper*/
                                     /* will be consulted to get the  */
                                     /* actual port of the remote     */
                                     /* program. Constant AF_INET is  */
                                     /* defined in <sys/socket.h> and */
                                     /* structure in_addr defined in  */
                                     /* <netinet/in.h>                */
      if ((client=clntudp_create(&server_addr,0x20000001,1, /*create a rpc */
                      pertry_timeout,&sock))==NULL)   /*client handle*/
      {                                               /* for the remote*/
         printf("Can't create RPC handle\n");         /* program.      */
         printf("Please try again later.\n");
         exit(0);
      }
      total_timeout.tv_sec=30;   /*  the total timeout interval of the     */
      total_timeout.tv_usec=0;  /*  rpc call in seconds and microseconds. */
                                /*  Since this timeout value is 5 times   */
                                /*  larger than the per try timeout value */
                                /*  so there may have a maximum of 5 tries.*/
      reply=(struct result *)malloc(sizeof(struct result));
      if ((clnt_call(client,2,xdr_request,&example,          /* this is */
           xdr_returntype,reply,total_timeout))!=RPC_SUCCESS)  /* a micro */
      {                                                /* which calls the */
         printf("The remote server is either");        /* remote procdure */
         printf(" busy or down.  Please try again ");  /* associated with */
         printf("later.\n");                           /* the client handle*/
         exit(0);                                      /* created above.  */
      }
      clnt_destroy(client);      /* deallocate the space assocated */
                                 /* the client handle.        */
      strcpy(command,"rsh bylands kill -19 "); /* restart the child process*/
      itos(proc_str,reply->childproc_id);      /* of the server's "system  */
      strcat(command,proc_str);                /* co-ordinator".           */
      system(command);
      return (reply);
}


/* The main body for the first experiment
   which tries to top an empty stack.  */

main()
{request example1;
 struct result *reply;
```

```
/* This XDR routine returns a structure, which contains
   the result of the user's request, to the user's calling process. */

bool_t xdr_returntype(xdrsp,ptr)
XDR *xdrsp;
struct result *ptr;
{
   if (!xdr_int(xdrsp,&ptr->childproc_id))
     return(FALSE);
   if (!xdr_int(xdrsp,&ptr->status))
     return(FALSE);
   if (!xdr_union(xdrsp,&ptr->utype,&ptr->uval,u_tag_arms,NULL))
     return(FALSE);
   return(TRUE);
}


/* This XDR routine (de)serializes all the
   information about the user's request.     */

bool_t xdr_request(xdrsp,ptr)
XDR *xdrsp;
request *ptr;
{int i;
 int maxapp=req_applength+2;
 int maxcmd=req_cmdlength+2;
 int maxdb=req_dblength+2;
 int maxentry=req_entrylength+2;
 int maxdata=req_datalength+2;
 int onebyte=1;
    for (i=0; i<maxapp; i++)
    { if (!xdr_bytes(xdrsp,ptr->application[i],&onebyte,onebyte))
        return(FALSE);   }
    for (i=0; i<maxcmd; i++)
    { if (!xdr_bytes(xdrsp,ptr->command[i],&onebyte,onebyte))
        return(FALSE);  }
    for (i=0; i<maxdb; i++)
    {  if (!xdr_bytes(xdrsp,ptr->dbname[i],&onebyte,onebyte))
         return(FALSE);    }
    for (i=0; i<maxentry; i++)
    { if (!xdr_bytes(xdrsp,ptr->entry[i],&onebyte,onebyte))
        return(FALSE);    }
    for (i=0; i<maxdata; i++)                                  /* vary with the */
    { if (!xdr_bytes(xdrsp,ptr->data[i],&onebyte,onebyte))    /* nature of the */
        return(FALSE);    }                                    /* application.*/
    return(TRUE);
}


/* This XDR routine is constructed particularly for
   the access manager to accepts ID information from
   caller processes.                                  */

bool_t xdr_idinfo(xdrsp,ptr)
XDR *xdrsp;
id_info *ptr;
{int i, maxsize=maxhostname;
 int onebyte=1;
    for (i=0; i<maxsize; i++)
    { if (!xdr_bytes(xdrsp,ptr->hostname[i],&onebyte,onebyte))
        return(FALSE);  }
    if (!xdr_int(xdrsp,&ptr->id))
      return(FALSE);
    return(TRUE);
}


/* This routine invokes the queue_manager of the "system co-ordinator"
   at the server site so as to register this client process on the
   server's FIFO queue.                                               */

bool_t callrpc_queue_manager(process_ID)
id_info process_ID;
{
   struct hostent *hp;       /* structure hostent defined in <netdb.h> */
   struct timeval pertry_timeout,total_timeout;  /* structure timeval defined
                                                    in <time.h>    */
   struct sockaddr_in server_addr;    /* structure sockaddr_in defined
                                           in <netinet/in.h>        */
   int sock= RPC_ANYSOCK;             /*  constant RPC_ANYSOCK defined in
                                          <rpc/svc.h> which is automatically
                                          included by the file <rpc/rpc.h>.
                                          The purpose of this variable is to
                                          ask the kernel to choose the most
                                          appropriate socket for establishing
                                          the rpc connection.       */
   register CLIENT *client;

         if ((hp=gethostbyname(server_machine))==NULL)
         {
           printf("Can't get address for remote machine %s\n",server_machine);
           return(FALSE);
         }
         pertry_timeout.tv_sec=1;      /*  timeout interval for each rpc    */
         pertry_timeout.tv_usec=0;     /*  call in seconds and microseconds */
         bcopy(hp->h_addr,(struct in_addr *)&server_addr.sin_addr,
               hp->h_length);             /* construct Internet address of */
         server_addr.sin_family=AF_INET;  /* the server by putting the name*/
         server_addr.sin_port=0;          /* family address and port number*/
                                          /* of the host. If the port number*/
                                          /* is 0, then the remote portmapper*/
                                          /* will be consulted to get the  */
                                          /* actual port of the remote     */
                                          /* program. Constant AF_INET is  */
                                          /* defined in <sys/socket.h> and */
```

```
   char command[req_cmdlength+2];     /* to satisfy the syntax of a string in*/
   char dbname[req_dblength+2];       /* the language PS-algol.  Also, the   */
   char entry[req_entrylength+2];     /* definition of the data items can be */
   char data[req_datalength+2];       /* changed subject to the purpose of   */
} request;                            /* the application.                    */


/* the structure below is used to store the result
        after the user's request is committed      */

enum uniontype {  val=1, error=2, dontcare=3 } ;
   /* enumeration type specifications of the union  */
   /* inside the structure result below.*/

/* A typedef construct can also be used for the following C structure,
   however no matter which way is used, the current compiler always
   gives an enumeration type clash warning despite that the program
   can still run successfully.                                  */

struct result
{ enum uniontype   utype;    /* the union's discriminant which aims to
                                   select the arms of the union.      */
    int childproc_id;
    bool_t  status;         /*  1 means success and 0 means fail  */
    union
    {
       char value[maxlength]; /* this definition may change subject
                                  to the nature of the application.  */
       char *errormsg; /* assumed to have a maximum of 255 characters */
       char *dont; /* this is just an end-marker of the union */
    } uval;
} ;



/* This routine takes in an empty string and an integer, and
   then it converts the integer into the corresponding ASCII string  */

itos(str,intnb)
char str[maxdigits];
int intnb;
{ int tmp,quot,rem,pos;
  char s[maxdigits],*value;
  tmp= -1;
  do
  {
    ++tmp;
    quot=intnb/10;
    rem=intnb%10;
    switch (rem)
    {
       case 0: s[tmp]='0';
               break;
       case 1: s[tmp]='1';
               break;
       case 2: s[tmp]='2';
```



```
               break;
       case 3: s[tmp]='3';
               break;
       case 4: s[tmp]='4';
               break;
       case 5: s[tmp]='5';
               break;
       case 6: s[tmp]='6';
               break;
       case 7: s[tmp]='7';
               break;
       case 8: s[tmp]='8';
               break;
       case 9: s[tmp]='9';
    }
    intnb=quot;
  } while (quot!=0);
  for (pos=0; tmp!= -1; ++pos)
  {
      value= &s[tmp--];
      str[pos]= *value;
  }
}


/* The following two XDR routines and a C structure are required
   for the (de)serialization of the union part of the structure
   that is going to be returned to this.                        */

bool_t xdr_error(xdrsp,ptr)
XDR *xdrsp;
char *ptr;
{
   if (!xdr_string(xdrsp,&ptr,255))
     return(FALSE);
   return(TRUE);
}

bool_t xdr_value(xdrsp,ptr)
XDR *xdrsp;
char *ptr;
{int i;
 int onebyte=1;
 int maxsize=maxlength;
    for (i=0; i<maxsize; i++)
    {  if (!xdr_bytes(xdrsp,ptr[i],&onebyte,onebyte))
          return(FALSE);    }
    return(TRUE);
}

struct xdr_discrim u_tag_arms[3] =    /* the definition of structure    */
{                                     /* xdr_discrim is located in      */
   { val, xdr_value},                 /* the header file <rpc/rpc.h>    */
   { error, xdr_error},
   { dontcare , NULL}
};
```

```
      let push.op=stack.table(push)
      let IOfile=open(datafile,read.mode)
      push.op(IOfile)
      let committed=commit()
        !and the rest is as before
    end
```

(2) In case of the data bank example, the following modification are used instead:

(a) delete the two part involving singlewriteop and single.readop

(b) Omit the structures, student.record, student.address and student.course

(c) For the four operations: insert.op, delete.op, search.op, and updat.op, replacing all the string arguments to an argument of type "file" such as the one shown in (1) above.


```c
/* This module is constructed for the first stage testing, that is all
   the experiments mentioned in section 7.2.1.3. of the main thesis, of
   the stack example using the structured remote data access method.
   Since the C program code for these experiments is mostly the same except
   that they have a different main program body, there is no point to
   present the same declaration part so many times and hence the main program
   bodies for each of these experiments are shown at the end of this program
   listing with some of them inside the comments.    */


#include <stdio.h>
#include <string.h>
#include <rpc/rpc.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netdb.h>
#include <netinet/in.h>

#define server_machine "bylands"
#define client_machine "ws_lodge1"


#define maxhostname 20
#define pathlength 80
#define maxlength 20
#define maxdigits 11
#define cmdlength 100
#define errorlength 255
#define req_applength 20
#define req_cmdlength 10
#define req_dblength 20
#define req_entrylength 30
#define req_datalength 20


/* define boolean variables */
#define bool_t int
#define TRUE 1
#define FALSE 0


/*  Identification details of the client process */

typedef struct id_info
{
    char hostname[maxhostname];
    int id;
} id_info;


typedef struct request
{
   char application[req_applength+2]; /* the extra two bytes are required in */
```

```
                    begin
                     let IOfile:=open(datafile,
                                      write.mode)
                     output IOfile,1
                     close(IOfile)
                    end
                    else
                    begin
                      let IOfile:=open(datafile,
                                       write.mode)
                      output IOfile,0
                      output IOfile,
                      committed(error.explain),
                      "'n"
                      close(IOfile)
                    end
                  end
       cmd="empty" :begin
                    let empty.op=stack.table(empty)
                    let return.value:=empty.op
                    let committed=commit()
                    if committed=nil then
                    begin
                      let IOfile=open(datafile,
                                      write.mode)
                      output IOfile,1
                      output IOfile,return.value,
                      "'n"
                      close(IOfile)
                    end
                    else
                    begin
                      let IOfile=open(datafile,
                                      write.mode)
                      output IOfile,0
                      output IOfile,return.value,
                      "'n"
                      close(IOfile)
                    end
                   end
application="students.data.bank" : begin
                  let cmd:=reads(IOfile)
                  let dbname=reads(IOfile)
                  let entry=reads(IOfile)
                  close(IOfile)
                  let lib:=open.database(dbname,passwd,
                                         "read")
                  let db.table=s.lookup(entry,lib)
                  case true of
                   cmd="insert": begin
                          let insert.op=db.table(insert)
                          insert.op()
                                 end
                   cmd="delete": begin
                          let delete.op=db.table(delete)
                          let IOfile=open(cmdfile,
```



```
                                  read.mode)
                   let name:=reads(IOfile)
                   close(IOfile)
                   delete.op(name)
                          end
            cmd="search": begin
                    let search.op=db.table(search)
                    let IOfile=open(cmdfile,
                                    read.mode)
                    let name=reads(IOfile)
                    close(IOfile)
                    serach.op(name)
                           end
            cmd="update": begin
                    let update.op=db.table(update)
                    let IOfile=open(cmdfile,
                                    read.mode)
                    let name:=reads(IOfile)
                    close(IOfile)
                    update.op(name)
                           end
            cmd="single.read": begin
                    let single.readop=
                        db.table(single.read)
                    let IOfile=open(cmdfile,
                                    read.mode)
                    let name=reads(IOfile)
                    let field=reads(IOfile)
                    close(IOfile)
                    single.readop(name,field)
                                end
            cmd="single.write": begin
                    let single.writeop=
                        db.table(single.write)
                    let IOfile=open(cmdfile,
                                    read.mode)
                    let name:=reads(IOfile)
                    let field=reads(IOfile)
                    close(IOfile)
                    single.writeop(name,field)
                                 end
            default: let dummy="dont know"
   default: let query="not implemented yet."


!End of the program


!For the un-structured approach, another version of the above program is
!provided which has the following alterations:

  (1) In case of the stack example, replacing all the string variable by
      a file descriptor, eg. when cmd="push", the following compound
      statement is used:

         begin
```

```
! The following PS-algol program is used to access the abstract objects
! stored in the PS-algol database named "utility" by means of the
! structured approach.


!Specification of the string stack which have the same structure
!of those shown in the PS-algol module utility.class.lib.

structure stack.component(string symbol; pntr next)

structure stack.package(proc(string)push; proc(->string)top;
                        proc()pop; proc(->string)empty)


! Specification of the object students.data.bank

structure student.record(string nmae,sex; int age; pntr address,course,other)

structure student.address(int house.nb; string street,town; pntr next.address)

structure student.course(string department; int year)


structure student.info.package(proc()insert; proc(string)delete;
                               proc(string)search; proc(string)update;
                               proc(string,string)single.readinfo;
                               proc(string,string)single.writeinfo)



! define useful variables and constants

let cmdfile="/usr/lib/ps/dbs/cmdfile"
let datafile="/usr/lib/ps/dbs/datafile"

let read.mode=0
let write.mode=1
let passwd="ok"


! Main program body

let IOfile:=open(cmdfile.read.mode)
let application:=reads(IOfile)

case true of
   application="StringStack" : begin
                                 let cmd:=reads(IOfile)    ! the file "cmdfile"
                                 let dbname:=reads(IOfile)
                                 let entry:=reads(IOfile)
                                 close(IOfile)
                                 let lib:=open.database(dbname,passwd,"read")

                                 ! Now try to get access the stack according to
                                 ! to the command
```



```
let stack.table:=s.lookup(entry,lib)
case true of
  cmd="push" : begin
                 let IOfile:=open(datafile,
                                  read.mode)
                 let data:=reads(IOfile)
                 close(IOfile)
                 let push.op=stack.table(push)
                 push.op() !perform operation
                 let committed=commit()
                 if committed=nil then
                 begin
                   let IOfile:=open(datafile,
                                    write.mode)
                   output IOfile,1
                   close(IOfile)
                 end
                 else
                 begin
                  let IOfile:=open(datafile,
                                   write.mode)
                   output IOfile,0
                   output IOfile,
                   committed(error.explain),
                   "'n"
                   close(IOfile)
                  end
              end
  cmd="top" : begin
                let top.op=stack.table(top)
                let return.data=top.op()
                let committed=commit()
                if committed=nil then
                begin
                  let IOfile=open(datafile,
                                  write.mode)
                  output IOfile,1
                  output IOfile,return.data
                  close(IOfile)
                end
                else
                begin
                 let IOfile:=open(datafile,
                                  write.mode)
                  output IOfile,0
                  output IOfile,
                  committed(error.explain),
                  "'n"
                  close(IOfile)
                 end
              end
  cmd="pop" : begin
                let pop.op=stack.table(pop)
                pop.op()
                let committed=commit()
                if committed=nil then
```

```
    let datafile:=open(IOfile,write.mode)
    output datafile,1
    close(datafile)
  end
  else begin
         let datafile:=open(IOfile,write.mode)
         output datafile,0
         output datafile,committed(error.explain),"'n"
         close(datafile)
       end
end


let search.info=proc(string name)
begin
  let db:=open.database.read()
  let address.list:=s.lookup("address.list.by.name",db)
  if address.list=nil do
  begin
    let datafile:=open(IOfile,write.mode)
    output datafile, 0
    output datafile, "No such record yet'n"
    close(datafile)
    abort
  end

  let person=s.lookup(name,address.list)     ! person is a file
  if person=nil then                         ! descriptor this time
  begin
    let datafile:=open(IOfile,write.mode)
    output datafile,0
    output datafile, "No such person on the list yet'n"
    close(datafile)
  end
  else begin
         let datafile=open(IOfile,write.mode)
         while ~eoi(person) do               !copy information
         begin                               !of the student record
           let onebyte=read.byte(person)     !into the file "datafile"
           output datafile,onebyte           !byte by byte.
         end
         close(datafile)
       end
end


let update.info=proc(string name)
begin
   let db=open.database.write()
   let address.list:=s.lookup("address.list.by.name",db)
   if address.list=nil do
   begin
     let datafile:=open(IOfile,write.mode)
     output datafile, 0
     output datafile, "No such record yet'n"
     close(datafile)
     abort
   end


   let person=s.lookup(name,address.list)
   if person=nil then
   begin
     let datafile:=open(IOfile,write.mode)
     output datafile,0
     output datafile, "No such person on the list yet'n"
     close(datafile)
   end
   else begin
          let datafile=open(IOfile,2)
          s.enter(name,address.list,datafile)
          let committed=commit()
          if committed=nil then
          begin
            let datafile:=open(IOfile,write.mode)
            output datafile,1
            close(datafile)
          end
          else begin
                 let datafile:=open(IOfile,write.mode)
                 output datafile,0
                 output datafile,committed(error.explain),"'n"
                 close(datafile)
               end
        end
end
```

```
                          s.enter(person(addess)(houde.nb).address.list,
                                  new.data)
                        end
          field="street" : begin
                        let new.data=reads(datafile)
                        s.enter(person(address)(street),address.list,
                                new.data)
                      end
          field="town" : begin
                      let new.data=reads(datafile)
                      s.enter(person(address)(town),address.list,new.data)
                    end
          field="department" : begin
                          let new.data=reads(datafile)
                          s.enter(person(course)(department),
                                  address.list,new.data)
                        end
          field="year" : begin
                      let new.data=readi(datafile)
                      s.enter(person(course)(year),address.list,new.data)
                    end
          default : let new.data="not known field"
          close(datafile)
          let committed=commit()
          if committed=nil then
          begin
            datafile:=open(IOfile,write.mode)
            output datafile,1
            close(datafile)
          end
     end
  end

  !Collect all the components of the onject student.data.bank

  student.info.package(insert.info,delete.info,search.info,update.info,
                       single.readinfo,single.writeinfo)

end


!***************************************************
!Put all the objects into the database 'utility'.*
!***************************************************

let lib:=open.database("utility","ok","write")
if lib is error.record do
begin
  lib:=create.database("utility","ok")
  lib:=open.database("utility","ok","write")
  if lib is error.record do
  begin
    write "Cannot open database because ",lib(error.explain),"'n"
    write "The program is aborted. 'n"
    abort
  end
```

```
end

!define the names of the abstract objects that are to be stored in the database
let application1=stack.pack()
let application2=student.info.pack()

!Enter all the classes into the database 'utility'.
s.enter("StringStack",lib,application1)
s.enter("students.data.bank",lib,application2)

!Check that the input process is successed.
let committed=commit()
if committed=nil then write "The objects have put into the database utility'n"
                  else write "Cannot put into the database because ",
                             committed(error.explain),"'n"


!End of the program.


! In case of the un-structured approach, the following
  modifications are required:


  (a) For the stack example, the appearence of the type definition string
      is replaced by "file" which means all the data will be treated as
      rows of bytes.

  (b) For the students data bank example, the following amendments are made:

      (1) Omit the two procedures single.readinfo and single.writeinfo as they
          could not performed using the un-structured approach.

      (2) Since no structure is imposed  on a student record, the three
          structures: student.record, student.address and student.course
          can be omitted.

      (3) The three procedures insert.info, search.info and update.info are
          modified as follows:


          let insert.info=proc(string name)   ! Note that this time this
          begin                               ! procedure requires a string
            let db:=open.database.write()     ! parameter as an ID hint
            let address.list:=s.lookup("address.list.by.name",lib)
            if address.list=nil do
            begin
              addrress.list:=table()
              s.enter("address.list.by.nmae",db,address.list)
            end
            let datafile:=open(IOfile,2)   ! 2 means read/write mode
            s.enter(name,address.list,datafile)
            let committed=commit()
            if committed=nil then
            begin
```

```
            output datafile,0
            output datafile,committed(error.explain),"'n"
            close(datafile)
          end
      end
end

!**********************************************
! Procedure to read a single field of a record*
!**********************************************
let single.readinfo=proc(string student,field)
begin
  let db:=open.database.read()
  let address.list:=s.lookup("address.list.by.name",db)
  if address.list=nil do
  begin
    let datafile:=open(IOfile,write.mode)
    output datafile, 0
    output datafile, "No such record yet'n"
    close(datafile)
    abort
  end

  let person=s.lookup(student,address.list)
  if person=nil then
  begin
    let datafile:=open(IOfile,write.mode)
    output datafile,0
    output datafile, "No such person on the list yet'n"
    close(datafile)
  end
  else begin
         case true of
            field="name" : begin
                             let result=person(name)
                           end
            field="sex" : begin
                            let result=person(sex)
                          end
            field="age" : begin
                            let result=person(age)
                          end
            field="house_nb" : begin
                                 let result=person(address)(house.nb)
                               end
            field="street" : begin
                               let result=person(address)(street)
                             end
            field="town" : begin
                             let result=person(address)(town)
                           end
            field="department" : begin
                                   let result=person(course)(department)
                                 end
            field="year" : begin
                             let result=person(course)(year)
                          end
            default : let result="not known field"
            let committed=commit()
            if committed=nil then
           begin
             let datfile:=open(IOfile,write.mode)
             output datafile,1
             output datafile,result
             close(datafile)
           end
           else begin
                  let datafile:=open(IOfile,write.mode)
                  output datafile.0
                  output datafile,committed(error.explain)
                  close(datafile)
                end
       end
end


!********************************************************
!Procedure to overwrite a particular field of a record *
!********************************************************
let single.writeinfo=proc(string student,field)
begin
  let db:=open.database.read()
  let address.list:=s.lookup("address.list.by.name",db)
  if address.list=nil then
  begin
    let datafile:=open(IOfile,write.mode)
    output datafile, 0
    output datafile, "No such record yet'n"
    close(datafile)
    abort
  end

  let person=s.lookup(student,address.list)
  if person=nil then
  begin
    let datafile:=open(IOfile,read.mode)
    case true of
       field="name" : begin
                        let new.data=reads(datafile)
                        s.enter(person(name),address.list,new.data)
                      end
       field="sex" : begin
                       let new.data=reads(datafile)
                       s.enter(person(sex),address.list,new.data)
                     end
       field="age" : begin
                       let new.data=readi(datafile)
                       s.enter(person(age),address.list,new.data)
                     end
       field="house_nb" : begin
                            let new.data=readi(datafile)
```

```
  let db=open.database.write()
  let address.list:=s.lookup("address.list.by.name",db)
  if address.list=nil do
  begin
     let datafile:=open(IOfile,write.mode)
     output datafile, 0
     output datafile, "No such record yet'n"
     close(datafile)
     abort
   end

   let person:=s.lookup(student,address.list)
   if person=nil then
   begin
     let datafile:=open(IOfile,write.mode)
     output datafile,0
     output datafile, "No such person on the list yet'n"
     close(datafile)
   end
   else begin
          s.enter(student,address.list,nil)
          let committed=commit()
          if committed=nil then
          begin
            let datafile:=open(IOfile,write.mode)
            output datafile,1
            close(datafile)
          end
          else begin
                 let datafile:=open(IOfile,write.mode)
                 output datafile,0
                 output datafile,committed(error.explain),"'n"
                 close(datafile)
                end
        end
end

!******************************************
!Procedure to search for a student's record*
!******************************************
let search.info=proc(string student)
begin
  let db:=open.database.read()
  let address.list:=s.lookup("address.list.by.name",db)
  if address.list=nil do
  begin
    let datafile:=open(IOfile,write.mode)
    output datafile, 0
    output datafile, "No such record yet'n"
    close(datafile)
    abort
  end

  let person:=s.lookup(student,address.list)
  if person=nil then
  begin
```

```
    let datafile:=open(IOfile,write.mode)
    output datafile,0
    output datafile, "No such person on the list yet'n"
    close(datafile)
  end
  else begin
         let datafile:=open(IOfile,write.mode)
         output datafile,1
         output datafile, person(name),"'n"
         output datafile, person(sex),"'n"
         output datafile, person(age):3,"'n"
         output datafile, person(address)(house.nb),"'n"
         output datafile, person(address)(street),"'n"
         output datafile, person(address)(town),"'n"
         output datafile, person(course)(department),"'n"
         output datafile, person(course)(year),"'n"
         close(datafile)
       end
end

!**********************************************
!Procedure to update a student's whole record*
!**********************************************
let update.info=proc()
begin
  let db:=open.database.write()
  let address.list:=s.lookup("address.list.by.name",db)
  if address.list=nil do
  begin
    let datafile:=open(IOfile,write.mode)
    output datafile, 0
    output datafile, "No such record yet'n"
    close(datafile)
    abort
  end

  let person:=s.lookup(student(name),address.list)
  if person=nil then
  begin
    let datafile:=open(IOfile,write.mode)
    output datafile,0
    output datafile, "No such person on the list yet'n"
    close(datafile)
  end
  else begin
         let new.info=enter.student.info()
         s.enter(student,address.list,new.info)
         let committed=commit()
         if committed=nil then
         begin
           let datafile:=open(IOfile,write.mode)
           output datafile,1
           close(datafile)
         end
         else begin
                let datafile:=open(IOfile,write.mode)
```

```
!************************************************************
!Procedure to open the database for writing students' information*
!************************************************************
let open.database.write=proc(->pntr)
begin
  let database:=open.database( "utility","ok","write")   !pre-defined
  if database is error.record do                         !location
  begin
    let datafile:=open(IOfile,write.mode)
    output datafile,0       ! indicate the status of the operation
    output datafile,database(error.explain),"'n"  ! generate an error message
    close(datafile)
    abort
  end
  database   ! return the database if no error
end

!************************************************************
!Procedure to open the database for reading students' information*
!************************************************************
let open.database.read=proc(->pntr)
begin
  let database=open.database( "utility","ok","read")
  if database is error.record do
  begin
    let datafile:=open(IOfile,write.mode)
    output datafile,0
    output datafile,database(error.explain),"'n"
    close(datafile)
  end
  database
end

!************************************************
!Procedure to read in the student's information *
!************************************************
let enter.student.info=proc(->pntr)
begin
  let datafile=open(IOfile,read.mode)

  let person.name=reads(datafile)
  let person.age=readi(datafile)
  let person.sex:=reads(datafile)
  let person.house.nb=readi(datafile)
  let person.street=reads(datafile)
  let person.town=reads(datafile)
  let person.dept=reads(datafile)
  let person.year=readi(datafile)

  close(datafile)

  !Construct the student's address.
  let person.address=student.address(person.house.nb,person.street,
                                     person.town,nil)
```



```
  !Construct the student's course
  let person.course=student.course(person.dept,person.year,nil)


  !Construct the student's full record.
  let person.record=student.record(person.name,person.sex,person.age,
                                   person.address,person.course,nil)


  !Return the student's record
  person.record
end


!The internal implementation of the operations of the class are defined:

!******************************************
!Procedure to insert student's information*
!******************************************
let insert.info=proc()
begin
  let db=open.database.write()
  let address.list:=s.lookup("address.list.by.name",db)

  !If the database is nil then it is the first entry
  if address.list=nil do
  begin
    address.list:=table()
    s.enter("address.list.by.name",db,address.list)
  end

   !Enter the student's information.
   let student=enter.student.info()

   !Enter the student's record into the database
   s.enter(student(name),address.list,student)

   !Commit the entering process
   let committed=commit()
   if committed=nil then
   begin
     let datafile:=open(IOfile,write.mode)
     output datafile,1
     close(datafile)
   end
   else begin
          let datafile=open(IOfile,write.mode)
          output datafile,0
          output datafile,committed(error.explain),"'n"
          close(datafile)
        end

end

let delete.info=proc(string student)
begin
```

```
! This is a PS-algol module which stores the internal representation
! of the two abstract objects: StringStack and student.data.bank., in
! a PS-algol  dtaabase named"utility".  Two versions of this module are
! implemented.  The first version is for the structured remote data access
! approach while the other one is for the un-structured appraoch.  Only
! the former will be shown in full here because the latter version is very
! similar to the former with just a few modifications which will be given
! at the end of this listing.


! *********************************************************************
! This part stores the internal representation of object "StringStack"*
! *********************************************************************

structure stack.components(string symbol, pntr next)

structure stack.package(proc(string)push; proc(->string)top;
                        proc()pop; proc(->string)empty)

let stack.pack=proc(->pntr)
begin
  let stack:=nil         ! initialize the stack

  let push.stack=proc(string item)
  begin
    stack:=stack(item,stack)
  end

  let top.stack=proc(->string)
  begin
    let status:="empty"
    let return.value="empty"

    status:=empty.stack()
    if status="ok" then
    begin
      return.value:=stack(symbol)
      return.value
    end
    else status
  end

  let pop.stack=proc()
  begin
    let status:="empty"
    status:=empty.stack()
    if status="ok" do
      stack:=stack(next)
  end

  let empty.stack=proc(->string)
  begin
    let isempty="stack is empty"
    let notempty="ok"

    if stack=nil then isempty
```



```
               else notempty
  end


  !collect operations of the object StringStack

   stack.package(push.stack,top.stack,pop.stack,empty.stack)

end

!***************************************************************************
!This part stores the internal representation of the object student.data.bank  *
!***************************************************************************

!Specification of the object

structure student.info.package(proc()insert; proc(string)delete;
                               proc(string)search; proc(string)update;
                               proc(string,string)single.read;
                               proc(string,string)single.write)


!Construction of the object

let student.info.pack=proc(->pntr)
begin

  !The student record contains the following information:
  !name, sex, age, address of the student and finally the course attended
  !by the student.
  !All of these information is stored in 3 parts.  There are 2 reasons for
  !doing this.  First of all, it makes things simple so it is easiler to debug
  !the program.  Secondly, it allows extra information to be added on by
  !providing the additional pointer field in each of them.

  !The first part gives the overall structure of the record
  structure student.record(string name,sex; int age; pntr address,course,other)

  !The second part defines the contents of the address field of the record
  structure student.address(int house.nb; string street,town; pntr next.address)

  !The third part defines the contents of the course field of the record
  structure student.course(string department; int year; pntr next.course)

  !The following 3 procedures are local to the onject
  !which are used to open the database for I/O purposes.


  ! modes for IOfile
  let read.mode=0
  let write.mode=1

  !define the I/O file which is associated with the operations of this module.
  let IOfile="/usr/lib/ps/dbs/datafile"
```

```
            strcpy(process_ID.hostname,server_machine);
            process_ID.id=getpid();
            if (callrpc_queue_manager(process_ID)==TRUE)
            {
               strcpy(command,"kill -17 ");
               itos(proc_str,process_ID.id);
               strcat(command,proc_str);
               system(command);
               reply=(struct result *)malloc(sizeof(struct result));
               reply=callrpc_access_manager(tmp);
               if (reply->status==TRUE)
               {
                   printf("The top value of the stack is ");
                   printf("%s\n",reply->uval.value);
               }
               else
               {
                  printf("%s\n",reply->uval.errormsg);
                  exit(0);
               }
            }
            strcpy(example1.application,"\"StringStack\"");
            strcpy(example1.command,"\"pop\"");
            strcpy(example1.dbname,"\"utility\"");
            strcpy(example1.entry,"\"application1\"");
            strcpy(process_ID.hostname,server_machine);
            process_ID.id=getpid();
            if (callrpc_queue_manager(process_ID)==TRUE)
            {
              strcpy(command,"kill -17 ");
              itos(proc_str,process_ID.id);
              strcat(command,proc_str);
              system(command);
              reply=(struct result *)malloc(sizeof(struct result));
              reply=callrpc_access_manager(tmp);
              if (reply->status==TRUE)
              {
                printf("The top value of the stack is ");
                printf("%s\n",reply->uval.value);
              }
              else
              {
                 printf("%s\n",reply->uval.errormsg);
                 exit(0);
              }
            }
        }
        else
        {
            printf("%s\n",reply->uval.errormsg);
        }
      }
    } while (strcmp(message,emptystack)!=0);
}    */
```

```
/* This module is designated as the "system co-ordinator" of the
   DCSUNIX system for the first stage testing of the stack example
   using the structured remote data access method.              */


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

/* define limits */

#define maxhostname 20
#define pathlength 80
#define maxlength 20
#define maxdigits 11
#define cmdlength 100
#define errorlength 255
#define req_applength 20
#define req_cmdlength 10
#define req_dblength 20
#define req_entrylength 30
#define req_datalength 20


/* define boolean variables */
#define bool_t int
#define TRUE 1
#define FALSE 0

/* define locations of the two Unix files and the PS-algol program "service" */

#define cmdfile "/usr/lib/ps/dbs/cmdfile"
#define datafile "/usr/lib/ps/dbs/datafile"
#define service "/user/res/mscwct/psalgol/service.out"


/*  implementation of the server's FIFO queue */
int host_fildes[2];
int procnb_fildes[2];

int pid; /* refers to the system co-ordinator's
            child process ID number                */


/*  Identification details of the client process */

typedef struct id_info
{
   char hostname[maxhostname];
   int id;
} id_info;



/*   the overall format of the user's request  */
```

```
typedef struct request
{
   char application[req_applength+2]; /* the extra two bytes are required in */
   char command[req_cmdlength+2];      /* to satisfy the syntax of a string  */
   char dbname[req_dblength+2];        /* in the langauge PS-algol. Also, the*/
   char entry[req_entrylength+2];      /* definition of the data items can be*/
   char data[req_datalength+2];        /* changed subject to the purpose of  */
} request;                             /* the application.                   */


/* the structure below is used to store the result
        after the user's request is committed      */

enum uniontype {  val=1, error=2, dontcare=3 } ;
   /* enumeration type specifications of the union  */
   /* inside the structure result below.*/

/* A typedef construct can also be used for the following C structure,
   however no matter which way is used, the current compiler always
   gives an enumeration type clash warning despite that the program
   can still run successfully.                                */

struct result
{ enum uniontype   utype;    /* the union's discriminant which aims to
                                  select the arms of the union.     */
    int childproc_id;
    bool_t  status;         /*  1 means success and 0 means fail  */
    union
    {
       char value[maxlength]; /* this definition may change subject
                                 to the nature of the application.  */
       char *errormsg; /* assumed to have a maximum of 255 characters */
       char *dont; /* this is just an end-marker of the union */
    } uval;
} ;


/* This routine takes in an empty string and an integer, and then
   it converts the integer into the corresponding ASCII string  */

itos(str,intnb)
char str[maxdigits];
int intnb;
{ int tmp,quot,rem,pos;
  char s[maxdigits],*value;
  tmp= -1;
  do
  {
    ++tmp;
    quot=intnb/10;
    rem=intnb%10;
    switch (rem)
    {
       case 0: s[tmp]='0';
               break;
       case 1: s[tmp]='1';
               break;
       case 2: s[tmp]='2';
               break;
       case 3: s[tmp]='3';
               break;
       case 4: s[tmp]='4';
               break;
       case 5: s[tmp]='5';
               break;
       case 6: s[tmp]='6';
               break;
       case 7: s[tmp]='7';
               break;
       case 8: s[tmp]='8';
               break;
       case 9: s[tmp]='9';
    }
    intnb=quot;
  } while (quot!=0);
  for (pos=0; tmp!= -1; ++pos)
  {
     value= &s[tmp--];
     str[pos]= *value;
  }
}


/* This routine takes in the name of the machine of a client
   process, together with the ID number of the process in
   that machine, then it will send a signal to re-start that process.*/

restart_proc(hostname,proc_id)
char hostname[maxhostname];
int proc_id;
{ char command[cmdlength];
  char proc_str[maxdigits];
     strcpy(command,"rsh ");
     strcat(command,hostname);
     strcat(command," kill -19 ");
     itos(proc_str,pid);
     strcat(command,proc_str);
     system(command);
}


/* This manager puts all the caller processes's
   ID information into the server's FIFO queue.
   This routine will be registered as a RPC shortly */

queue_manager(client_info)
id_info *client_info;
{
```

```
   write(host_fildes[1],client_info->hostname,maxhostname);
   write(procnb_fildes[1],&client_info->id,sizeof(int));
}


/* This is a second RPC which sends all the information
   about a user's request to the two Unix files cmdfile
   and datafile accordingly, then those information will
   be consumed by the PS-algol "service" to carry out the
   user's request.                                    */

struct result *access_manager(request_info)
request *request_info;
{FILE *fp;
 char executeprog[pathlength];
 struct result *answer;

   fp=fopen(cmdfile,"w");                           /* transfer information to */
   fprintf(fp,"%s\n",request_info->application);   /* the file cmdfile in     */
   fprintf(fp,"%s\n",request_info->command);       /* order to execute the    */
   fprintf(fp,"%s\n",request_info->dbname);        /* client's request        */
   fprintf(fp,"%s\n",request_info->entry);
   fclose(fp);

   fp=fopen(datafile,"w");                         /* this part will have no  */
   fprintf(fp,"%s\n",request_info->data);          /* effect on file datafile */
   fclose(fp);                                     /* if the user's request   */
                                                   /* is in the read mode. */

   strcpy(executeprog,"psr");
   strcat(executeprog,service);
   system(executeprog);         /* carry out the user's request.*/

   answer=(struct result*)malloc(sizeof(struct result));
   answer->childproc_id=pid;    /* this part retruns the result of the request*/
   fp=fopen(datafile,"r");
   fscanf(fp,"%d\n",&answer->status);
   if (answer->status==TRUE)
   {  answer->utype=val;
      fscanf(fp,"%s\n",answer->uval.value);   }
   else
   {   answer->utype=error;
       fscanf(fp,"%s",answer->uval.errormsg); }
   fclose(fp);
   return(answer);
}


/* The following two XDR routines and a C structure are required
   for the (de)serialization of the union part of the structure
   that is going to be returned to the user's calling process. */

bool_t xdr_error(xdrsp,ptr)
XDR *xdrsp;
char *ptr;
{
   if (!xdr_string(xdrsp,&ptr,255))
     return(FALSE);
   return(TRUE);
}

bool_t xdr_value(xdrsp,ptr)
XDR *xdrsp;
char *ptr;
{int i;
 int onebyte=1;
 int maxsize=maxlength;
    for (i=0; i<maxsize; i++)
    {  if (!xdr_bytes(xdrsp,ptr[i],&onebyte,onebyte))
          return(FALSE);    }
    return(TRUE);
}

struct xdr_discrim u_tag_arms[3] =     /* the definition of structure    */
{                                      /* xdr_discrim is located in      */
   { val, xdr_value},                  /* the header file <rpc/rpc.h>    */
   { error, xdr_error},
   { dontcare , NULL}
};


/* This XDR routine returns a structure, which contains
   the result of the user's request, to the user's calling process. */

bool_t xdr_returntype(xdrsp,ptr)
XDR *xdrsp;
struct result *ptr;
{
  if (!xdr_int(xdrsp,&ptr->childproc_id))
    return(FALSE);
  if (!xdr_int(xdrsp,&ptr->status))
    return(FALSE);
  if (!xdr_union(xdrsp,&ptr->utype,&ptr->uval,u_tag_arms,NULL))
    return(FALSE);
  return(TRUE);
}


/* This XDR routine (de)serializes all the
   information about the user's request.     */

bool_t xdr_request(xdrsp,ptr)
XDR *xdrsp;
request *ptr;
{int i;
 int maxapp=req_applength+2;
 int maxcmd=req_cmdlength+2;
 int maxdb=req_dblength+2;
 int maxentry=req_entrylength+2;
 int maxdata=req_datalength+2;
 int onebyte=1;
```

```
    for (i=0; i<maxapp; i++)
    { if (!xdr_bytes(xdrsp,ptr->application[i],&onebyte,onebyte))
        return(FALSE);    }
    for (i=0; i<maxcmd; i++)
     { if (!xdr_bytes(xdrsp,ptr->command[i],&onebyte,onebyte))
         return(FALSE);   }
    for (i=0; i<maxdb; i++)
     {  if (!xdr_bytes(xdrsp,ptr->dbname[i],&onebyte,onebyte))
          return(FALSE);    }
    for (i=0; i<maxentry; i++)
     { if (!xdr_bytes(xdrsp,ptr->entry[i],&onebyte,onebyte))
         return(FALSE);    }
    for (i=0; i<maxdata; i++)                                 /* vary with the */
     { if (!xdr_bytes(xdrsp,ptr->data[i],&onebyte,onebyte))  /* nature of the */
         return(FALSE);    }                                  /* application.*/
    return(TRUE);
}


/* This XDR routine is constructed particularly for
   the access manager to accepts ID information from
   caller processes.                                  */

bool_t xdr_idinfo(xdrsp,ptr)
XDR *xdrsp;
id_info *ptr;
{int i, maxsize=maxhostname;
 int onebyte=1;
   for (i=0; i<maxsize; i++)
     { if (!xdr_bytes(xdrsp,ptr->hostname[i],&onebyte,onebyte))
         return(FALSE);   }
   if (!xdr_int(xdrsp,&ptr->id))
     return(FALSE);
   return(TRUE);
}


/* This is the main body of this module which comprises two
   infinite processes.  The first process, a child process of
   this module created by a fork system call, determines the
   next user of the DCSUNIX system whereas the second one,
   the parent process, is merely used for the registration of
   the two RPCs defined previously at the local host "bylands".  */

main()
{ int proc_id;
  char caller_hostname[maxhostname];
  char command[cmdlength];
  char proc_str[maxdigits];

     pipe(host_fildes);  /* sets up a FIFO queue using two Unix pipes  */
     pipe(procnb_fildes);
     pid=fork();        /* the process number of the child process */
     if (pid==0)        /* will be returned to the parent process  */
     {
        while(1==1)
        {
           read(host_fildes[0],caller_hostname, maxhostname);
           read(procnb_fildes[0],&proc_id,sizeof(int));
           restart_proc(caller_hostname,proc_id);
           pid=getpid();  /* the process number of the child process itself. */
           strcpy(command,"kill -17 ");
           itos(proc_str,pid);
           strcat(command,proc_str);
           system(command);
        }
     }
     else
     {
       registerrpc(0x20000001,1,1,queue_manager,xdr_idinfo,xdr_void);
       registerrpc(0x20000001,1,2,access_manager,xdr_request,xdr_returntype);
       svc_run();
     }
}
```

```
/* This module is designated as the "system co-ordinator" of the
   DCSUNIX system for the first stage testing of the students data
   bank example using the structured remote data access method.
   However, in order to perform all the experiments described in
   section 7.2.2.3. of the main thesis, different versions of this
   module are required.  To save time and space, only the version for the
   first experiment will be shown in full here.  */


#include <stdio.h>
#include <string.h>
#include <rpc/rpc.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netdb.h>
#include <netinet/in.h>

#define server_machine "bylands"
#define client_machine "ws_lodgel"


#define maxhostname 20
#define pathlength 80
#define maxlength 20
#define maxdigits 11
#define cmdlength 100
#define errorlength 255
#define req_applength 20
#define req_cmdlength 10
#define req_dblength 20
#define req_entrylength 30
#define namelen 30
#define streetlen 30
#define townlen 20
#define departlen 30


/* define boolean variables */
#define bool_t int
#define TRUE 1
#define FALSE 0


/*  Identification details of the client process */

typedef struct id_info
{
    char hostname[maxhostname];
    int id;
} id_info;


/* the general structure of a student record in the data bank. */

typedef struct student_record
{
  char name[namelen];
  int age;
  char sex;
  int house_nb;
  char street[streetlen];
  char town[townlen];
  char department[departlen];
  int year;
} student_record;


typedef struct request
{
   char application[req_applength+2]; /* the extra two bytes are required in */
   char command[req_cmdlength+2];     /* to satisfy the syntax of a string in*/
   char dbname[req_dblength+2];       /* the language PS-algol.  Also, the   */
   char entry[req_entrylength+2];     /* definition of the data items can be */
   char student[namelen+2];           /* changed subject to the purpose of   */
} request;                            /* the application.                    */


/* the structure below is used to store the result
        after the user's request is committed      */

enum uniontype {  val=1, error=2, dontcare=3 } ;
   /* enumeration type specifications of the union  */
   /* inside the structure result below.*/

/* A typedef construct can also be used for the following C structure,
   however no matter which way is used, the current compiler always
   gives an enumeration type clash warning despite that the program
   can still run successfully.                                  */

struct result
{ enum uniontype   utype;    /* the union's discriminant which aims to
                                  select the arms of the union.     */
    int childproc_id;
    bool_t  status;         /*  1 means success and 0 means fail  */
    union
    {
       student_record *value; /* this definition may change subject
                                  to the nature of the application.  */
       char *errormsg; /* assumed to have a maximum of 255 characters */
       char *dont; /* this is just an end-marker of the union */
    } uval;
} ;


/* This routine takes in an empty string and an integer, and
   then it converts the integer into the corresponding ASCII string  */

itos(str,intnb)
char str[maxdigits];
```

```c
int intnb;
{ int tmp,quot,rem,pos;
  char s[maxdigits],*value;
  tmp= -1;
  do
  {
    ++tmp;
    quot=intnb/10;
    rem=intnb%10;
    switch (rem)
    {
       case 0: s[tmp]='0';
               break;
       case 1: s[tmp]='1';
               break;
       case 2: s[tmp]='2';
               break;
       case 3: s[tmp]='3';
               break;
       case 4: s[tmp]='4';
               break;
       case 5: s[tmp]='5';
               break;
       case 6: s[tmp]='6';
               break;
       case 7: s[tmp]='7';
               break;
       case 8: s[tmp]='8';
               break;
       case 9: s[tmp]='9';
    }
    intnb=quot;
  } while (quot!=0);
  for (pos=0; tmp!= -1; ++pos)
  {
      value= &s[tmp--];
      str[pos]= *value;
  }
}

/* The following two XDR routines and a C structure are required
   for the (de)serialization of the union part of the structure
   that is going to be returned to this.                        */

bool_t xdr_error(xdrsp,ptr)
XDR *xdrsp;
char *ptr;
{
   if (!xdr_string(xdrsp,&ptr,255))
     return(FALSE);
   return(TRUE);
}

bool_t xdr_value(xdrsp,ptr)
XDR *xdrsp;
student_record *ptr;
{int i;
 int onebyte=1;
 int namelg=namelen;
 int streetlg=streetlen;
 int townlg=townlen;
 int departlg=departlen;
   for (i=0; i<namelg; i++)
   {  if (!xdr_bytes(xdrsp,ptr->name[i],&onebyte,onebyte))
        return(FALSE);   }
   if (!xdr_int(xdrsp,&ptr->age))
     return(FALSE);
   if (!xdr_bytes(xdrsp,&ptr->sex,&onebyte,onebyte))
     return(FALSE);
   if (!xdr_int(xdrsp,&ptr->house_nb))
     return(FALSE);
   for (i=0; i<streetlg; i++)
   {  if (!xdr_bytes(xdrsp,ptr->street[i],&onebyte,onebyte))
        return(FALSE);   }
   for (i=0; i<townlg; i++)
   {  if (!xdr_bytes(xdrsp,ptr->town[i],&onebyte,onebyte))
        return(FALSE);   }
   for (i=0; i<departlg; i++)
   {  if (!xdr_bytes(xdrsp,ptr->department,&onebyte,onebyte))
        return(FALSE);    }
   if (!xdr_int(xdrsp,&ptr->year))
     return(FALSE);
   return(TRUE);
}

struct xdr_discrim u_tag_arms[3] =    /* the definition of structure    */
{                                     /* xdr_discrim is located in      */
   { val, xdr_value},                 /* the header file <rpc/rpc.h>    */
   { error, xdr_error},
   { dontcare , NULL}
};


/* This XDR routine returns a structure, which contains
   the result of the user's request, to the user's calling process. */

bool_t xdr_returntype(xdrsp,ptr)
XDR *xdrsp;
struct result *ptr;
{
   if (!xdr_int(xdrsp,&ptr->childproc_id))
     return(FALSE);
   if (!xdr_int(xdrsp,&ptr->status))
     return(FALSE);
   if (!xdr_union(xdrsp,&ptr->utype,&ptr->uval,u_tag_arms,NULL))
     return(FALSE);
   return(TRUE);
}
```

```
/* This XDR routine (de)serializes all the
   information about the user's request.     */

bool_t xdr_request(xdrsp,ptr)
XDR *xdrsp;
request *ptr;
{int i;
 int maxapp=req_applength+2;
 int maxcmd=req_cmdlength+2;
 int maxdb=req_dblength+2;
 int maxentry=req_entrylength+2;
 int maxname=namelen+2;
 int onebyte=1;
    for (i=0; i<maxapp; i++)
    { if (!xdr_bytes(xdrsp,ptr->application[i],&onebyte,onebyte))
        return(FALSE);    }
    for (i=0; i<maxcmd; i++)
    { if (!xdr_bytes(xdrsp,ptr->command[i],&onebyte,onebyte))
        return(FALSE);  }
    for (i=0; i<maxdb; i++)
    {  if (!xdr_bytes(xdrsp,ptr->dbname[i],&onebyte,onebyte))
         return(FALSE);    }
    for (i=0; i<maxentry; i++)
    { if (!xdr_bytes(xdrsp,ptr->entry[i],&onebyte,onebyte))
        return(FALSE);    }
    for (i=0; i<maxname; i++)                            /* vary with the */
    { if (!xdr_bytes(xdrsp,ptr->student[i],&onebyte,onebyte))/* nature of the */
        return(FALSE);    }                              /* application.*/
    return(TRUE);
}


/* This XDR routine is constructed particularly for
   the access manager to accepts ID information from
   caller processes.                                */

bool_t xdr_idinfo(xdrsp,ptr)
XDR *xdrsp;
id_info *ptr;
{int i, maxsize=maxhostname;
 int onebyte=1;
    for (i=0; i<maxsize; i++)
    { if (!xdr_bytes(xdrsp,ptr->hostname[i],&onebyte,onebyte))
        return(FALSE);  }
    if (!xdr_int(xdrsp,&ptr->id))
      return(FALSE);
    return(TRUE);
}


/* This routine invokes the queue_manager of the "system co-ordinator"
   at the server site so as to register this client process on the
   server's FIFO queue.                                                 */

bool_t callrpc_queue_manager(process_ID)
id_info process_ID;
{
   struct hostent *hp;       /* structure hostent defined in <netdb.h> */
   struct timeval pertry_timeout,total_timeout;  /* structure timeval defined
                                                      in <time.h>   */
   struct sockaddr_in server_addr;    /* structure sockaddr_in defined
                                           in <netinet/in.h>          */
   int sock= RPC_ANYSOCK;             /*  constant RPC_ANYSOCK defined in
                                          <rpc/svc.h> which is automatically
                                          included by the file <rpc/rpc.h>.
                                          The purpose of this variable is to
                                          ask the kernel to choose the most
                                          appropriate socket for establishing
                                          the rpc connection.        */
   register CLIENT *client;

         if ((hp=gethostbyname(server_machine))==NULL)
         {
           printf("Can't get address for remote machine %s\n",server_machine);
           return(FALSE);
         }
         pertry_timeout.tv_sec=1;      /*  timeout interval for each rpc    */
         pertry_timeout.tv_usec=0;     /*  call in seconds and microseconds */
         bcopy(hp->h_addr,(struct in_addr *)&server_addr.sin_addr,
               hp->h_length);              /* construct Internet address of */
         server_addr.sin_family=AF_INET;   /* the server by putting the name*/
         server_addr.sin_port=0;           /* family address and port number*/
                                           /* of the host. If the port number*/
                                           /* is 0, then the remote portmapper*/
                                           /* will be consulted to get the  */
                                           /* actual port of the remote     */
                                           /* program. Constant AF_INET is  */
                                           /* defined in <sys/socket.h> and */
                                           /* structure in_addr defined in  */
                                           /* <netinet/in.h>                */
         if ((client=clntudp_create(&server_addr,0x20000001,1, /*create a rpc*/
                          pertry_timeout,&sock))==NULL)   /*client handle*/
         {                                                 /* for the remote*/
            printf("Can't create RPC handle\n");           /* program.      */
            return(FALSE);
         }
         total_timeout.tv_sec=1;   /*  the total timeout interval of the     */
         total_timeout.tv_usec=0;  /*  rpc in seconds and microseconds which */
                                   /*  has the same value as the timeout of */
                                   /*  each try and therefore only one call */
                                   /*  is issued each time.                 */
         if (clnt_call(client,1,xdr_idinfo,&process_ID,   /* this is a micro */
                 xdr_void,0,total_timeout)!=RPC_SUCCESS)/* which calls the */
         {                                              /* remote procedure*/
           printf("The remote server is either");       /* associated with */
           printf(" busy or down.  Please try again ");  /* the client      */
           printf("later.\n");                          /* handle created  */
           return(FALSE);                               /* above.          */
```

```
        }
        clnt_destroy(client);       /* deallocate the space assocated */
                                    /* with the client handle         */
        return(TRUE);
}


/* This routine invokes the access_manager of the "system co-ordinator"
   at the server site so as to access the required abstract object stored
   in the PS-algol database system.                               */

struct result *callrpc_access_manager(example)
request example;
{
   struct hostent *hp;       /* structure hostent defined in <netdb.h> */
   struct timeval pertry_timeout,total_timeout;  /* structure timeval defined
                                                   in <time.h>            */
   struct sockaddr_in server_addr;    /* structure sockaddr_in defined
                                           in <netinet/in.h>         */
   int sock= RPC_ANYSOCK;       /*  constant RPC_ANYSOCK defined in
                                    <rpc/svc.h> which is automatically
                                    included by the header file <rpc/rpc.h>.
                                    the purpose of this variable is to ask
                                    the kernel to choose the appropriate
                                    socket to establish the rpc connection.*/


   struct result *reply;

   char command[cmdlength];
   char proc_str[maxdigits];

   register CLIENT *client;

        if ((hp=gethostbyname(server_machine))==NULL)
        {
           printf("Can't get address for remote machine %s\n",server_machine);
           printf("Please try again later.\n");
           exit(0);
        }
        pertry_timeout.tv_sec=6;      /*  timeout interval for each rpc    */
        pertry_timeout.tv_usec=0;     /*  call in seconds and microseconds */
        bcopy(hp->h_addr,(struct in_addr *)&server_addr.sin_addr,
              hp->h_length);             /* construct Internet address of */
        server_addr.sin_family=AF_INET; /* the server by putting the name*/
        server_addr.sin_port=0;          /* family address and port number*/
                                         /* of the host. If port number  */
                                         /* is 0, then the remote portmapper*/
                                         /* will be consulted to get the  */
                                         /* actual port of the remote     */
                                         /* program. Constant AF_INET is  */
                                         /* defined in <sys/socket.h> and */
                                         /* structure in_addr defined in  */
                                         /* <netinet/in.h>                */
        if ((client=clntudp_create(&server_addr,0x20000001,1, /*create a rpc */
                       pertry_timeout,&sock))==NULL)    /*client handle*/
           {                                                        /* for the remote*/
              printf("Can't create RPC handle\n");                  /* program.     */
              printf("Please try again later.\n");
              exit(0);
           }
           total_timeout.tv_sec=30;   /*  the total timeout interval of the     */
           total_timeout.tv_usec=0;  /*  rpc call in seconds and microseconds. */
                                     /*  Since this timeout value is 5 times   */
                                     /*  larger than the per try timeout value */
                                     /*  so there may have a maximum of 5 tries.*/
           reply=(struct result *)malloc(sizeof(struct result));
           if ((clnt_call(client,2,xdr_request,&example,              /* this is */
                 xdr_returntype,reply,total_timeout))!=RPC_SUCCESS)  /* a micro */
           {                                                     /* which calls the */
              printf("The remote server is either");             /* remote procdure */
              printf(" busy or down.  Please try again ");  /* associated with */
              printf("later.\n");                                /* the client handle*/
              exit(0);                                           /* created above.   */
           }
           clnt_destroy(client);      /* deallocate the space assocated */
                                      /* the client handle.         */
           strcpy(command,"rsh bylands kill -19 "); /* restart the child process*/
           itos(proc_str,reply->childproc_id);      /* of the server's "system  */
           strcat(command,proc_str);                /* co-ordinator".           */
           system(command);
           return (reply);
}


/* The main body for the first experiment
   which tries to search for a particular record.  */

main()
{request example1;
 struct result *reply;
 id_info process_ID;
 char command[cmdlength];
 char proc_str[maxdigits];
   strcpy(example1.application,"\"student.data.bank\"");
   strcpy(example1.command,"\"search\"");
   strcpy(example1.dbname,"\"utility\"");
   strcpy(example1.entry,"\"application2\"");
   strcpy(example1.student,"\'Paul\"");
   strcpy(process_ID.hostname,server_machine);
   process_ID.id=getpid();
   if (callrpc_queue_manager(process_ID)==TRUE)
   {
     strcpy(command,"kill -17 ");    /* signal a sleep signal to itself*/
     itos(proc_str,process_ID.id);  /* and then waiting for service.  */
     strcat(command,proc_str);
     system(command);
     reply=(struct result *)malloc(sizeof(struct result));  /* this is the   */
     reply=callrpc_access_manager(example1);                /* re-start point*/
     if (reply->status==TRUE)
     {
```

```c
         printf("name : %s\n",reply->uval.value->name);
         printf("age  : %d\n",reply->uval.value->age);
         printf("sex  : %c\n",reply->uval.value->sex);
         printf("house number : %d\n",reply->uval.value->house_nb);
         printf("street : %s\n",reply->uval.value->street);
         printf("town : %s\n",reply->uval.value->town);
         printf("department : %s\n",reply->uval.value->department);
         printf("year : %s\n",reply->uval.value->year);
      }
      else
      {
         printf("%s\n",reply->uval.errormsg);
      }
   }
}


/* The other experiments can be performed in a similar fashion.
   The key idea is to change:

   (a) the last field of structure request according to the application,

   (b) the first field of the structure result,

   (c) the two routines xdr_request and xdr_value with respect to the
       change in (a) and (b) respectively.                */
```

```c
/* This module is designated as the "system co-ordinator" of the
   DCSUNIX system for the first stage testing of the students data
   bank example using the structured remote data access method.
   However, in order to perform all the experiments described in
   section 7.2.2.3. of the main thesis, different versions of this
   module are required.  To save time and space, only the version for the
   first experiment will be shown in full here; whereas the other versions
   will be shown in a simplified way at the end of this listing.        */


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

/* define limits */

#define maxhostname 20
#define pathlength 80
#define maxlength 20
#define maxdigits 11
#define cmdlength 100
#define errorlength 255
#define req_applength 20
#define req_cmdlength 10
#define req_dblength 20
#define req_entrylength 30
#define namelen 30
#define streetlen 30
#define townlen 20
#define departlen 30


/* define boolean variables */
#define bool_t int
#define TRUE 1
#define FALSE 0

/* define locations of the two Unix files and the PS-algol program "service" */

#define cmdfile "/usr/lib/ps/dbs/cmdfile"
#define datafile "/usr/lib/ps/dbs/datafile"
#define service "/user/res/mscwct/psalgol/service.out"


/*  implementation of the server's FIFO queue */
int host_fildes[2];
int procnb_fildes[2];

int pid; /* refers to the system co-ordinator's
            child process ID number              */


/*  Identification details of the client process */

typedef struct id_info
{
```

```c
   char hostname[maxhostname];
   int id;
} id_info;


/* the general structure of a student record in the data bank. */

typedef struct student_record
{
  char name[namelen];
  int age;
  char sex;
  int house_nb;
  char street[streetlen];
  char town[townlen];
  char department[departlen];
  int year;
} student_record;


/*   the overall format of the user's request  */

typedef struct request
{
   char application[req_applength+2]; /* the extra two bytes are required in */
   char command[req_cmdlength+2];      /* to satisfy the syntax of a string  */
   char dbname[req_dblength+2];        /* in the langauge PS-algol.          */
   char entry[req_entrylength+2];
   char student[namelen+2];   /* this is used as an ID hint of*/
} request;                    /* the student interested.     */


/* the structure below is used to store the result
        after the user's request is committed      */

enum uniontype {  val=1, error=2, dontcare=3 } ;
   /* enumeration type specifications of the union  */
   /* inside the structure result below.*/

/* A typedef construct can also be used for the following C structure,
   however no matter which way is used, the current compiler always
   gives an enumeration type clash warning despite that the program
   can still run successfully.                                  */

struct result
{ enum uniontype   utype;    /* the union's discriminant which aims to
                                  select the arms of the union.      */
    int childproc_id;
    bool_t  status;          /*  1 means success and 0 means fail  */
    union
    {
  .   student_record *value; /* this definition may change subject
                                to the nature of the application.  */
      char *errormsg; /* assumed to have a maximum of 255 characters */
      char *dont; /* this is just an end-marker of the union */
    } uval;
} ;


/* This routine takes in an empty string and an integer, and then
   it converts the integer into the corresponding ASCII string  */

itos(str,intnb)
char str[maxdigits];
int intnb;
{ int tmp,quot,rem,pos;
  char s[maxdigits],*value;
  tmp= -1;
  do
  {
    ++tmp;
    quot=intnb/10;
    rem=intnb%10;
    switch (rem)
    {
       case 0: s[tmp]='0';
               break;
       case 1: s[tmp]='1';
               break;
       case 2: s[tmp]='2';
               break;
       case 3: s[tmp]='3';
               break;
       case 4: s[tmp]='4';
               break;
       case 5: s[tmp]='5';
               break;
       case 6: s[tmp]='6';
               break;
       case 7: s[tmp]='7';
               break;
       case 8: s[tmp]='8';
               break;
       case 9: s[tmp]='9';
    }
    intnb=quot;
  } while (quot!=0);
  for (pos=0; tmp!= -1; ++pos)
  {
      value= &s[tmp--];
      str[pos]= *value;
  }
}


/* This routine takes in the name of the machine of a client
   process, together with the ID number of the process in
   that machine, then it will send a signal to re-start that process.*/
```

```
restart_proc(hostname,proc_id)
char hostname[maxhostname];
int proc_id;
{ char command[cmdlength];
  char proc_str[maxdigits];
     strcpy(command,"rsh ");
     strcat(command,hostname);
     strcat(command," kill -19 ");
     itos(proc_str,pid);
     strcat(command,proc_str);
     system(command);
}


/* This manager puts all the caller processes's
   ID information into the server's FIFO queue.
   This routine will be registered as a RPC shortly */

queue_manager(client_info)
id_info *client_info;
{
   write(host_fildes[1],client_info->hostname,maxhostname);
   write(procnb_fildes[1],&client_info->id,sizeof(int));
}


/* This is a second RPC which sends all the information
   about a user's request to the two Unix files cmdfile
   and datafile accordingly, then those information will
   be consumed by the PS-algol "service" to carry out the
   user's request.                                      */

struct result *access_manager(request_info)
request *request_info;
{FILE *fp;
 char executeprog[pathlength];
 struct result *answer;

   fp=fopen(cmdfile,"w");                          /* transfer information to */
   fprintf(fp,"%s\n",request_info->application);   /* the file cmdfile in     */
   fprintf(fp,"%s\n",request_info->command);       /* order to execute the    */
   fprintf(fp,"%s\n",request_info->dbname);        /* client's request        */
   fprintf(fp,"%s\n",request_info->entry);
   fprintf(fp,"%s\n",request_info->student);
   fclose(fp);

   strcpy(executeprog,"psr");
   strcat(executeprog,service);
   system(executeprog);         /* carry out the user's request.*/

   answer=(struct result*)malloc(sizeof(struct result));
   answer->childproc_id=pid;    /* this part retruns the result of the request*/
   fp=fopen(datafile,"r");
   fscanf(fp,"%d\n",&answer->status);
   if (answer->status==TRUE)
   {
      answer->utype=val;
      answer->uval.value=(student_record *)malloc(sizeof(student_record));
      fscanf(fp,"%s",(answer->uval.value)->name);
      fscanf(fp,"%d",(answer->uval.value)->age);
      fscanf(fp,"%c",(answer->uval.value)->sex);
      fscanf(fp,"%s",(answer->uval.value)->house_nb);
      fscanf(fp,"%s",(answer->uval.value)->street);
      fscanf(fp,"%s",(answer->uval.value)->town);
      fscanf(fp,"%s",(answer->uval.value)->department);
      fscanf(fp,"%d",(answer->uval.value)->year);
   }
   else
   {   answer->utype=error;
       fscanf(fp,"%s",answer->uval.errormsg); }
   fclose(fp);
   return(answer);
}


/* The following two XDR routines and a C structure are required
   for the (de)serialization of the union part of the structure
   that is going to be returned to the user's calling process. */

bool_t xdr_error(xdrsp,ptr)
XDR *xdrsp;
char *ptr;
{
   if (!xdr_string(xdrsp,&ptr,255))
     return(FALSE);
   return(TRUE);
}

bool_t xdr_value(xdrsp,ptr)
XDR *xdrsp;
student_record *ptr;
{int i;
 int onebyte=1;
 int namelg=namelen;
 int streetlg=streetlen;
 int townlg=townlen;
 int departlg=departlen;
   for (i=0; i<namelg; i++)
   {  if (!xdr_bytes(xdrsp,ptr->name[i],&onebyte,onebyte))
        return(FALSE);  }
   if (!xdr_int(xdrsp,&ptr->age))
     return(FALSE);
   if (!xdr_bytes(xdrsp,&ptr->sex,&onebyte,onebyte))
     return(FALSE);
   if (!xdr_int(xdrsp,&ptr->house_nb))
     return(FALSE);
   for (i=0; i<streetlg; i++)
   {  if (!xdr_bytes(xdrsp,ptr->street[i],&onebyte,onebyte))
        return(FALSE);  }
   for (i=0; i<townlg; i++)
```

```
   {  if (!xdr_bytes(xdrsp,ptr->town[i],&onebyte,onebyte))
         return(FALSE);   }
   for (i=0; i<departlg; i++)
   {  if (!xdr_bytes(xdrsp,ptr->department,&onebyte,onebyte))
         return(FALSE);    }
   if (!xdr_int(xdrsp,&ptr->year))
     return(FALSE);
   return(TRUE);
}

struct xdr_discrim u_tag_arms[3] =     /* the definition of structure    */
{                                      /* xdr_discrim is located in      */
   { val, xdr_value},                  /* the header file <rpc/rpc.h>    */
   { error, xdr_error},
   { dontcare , NULL}
};


/* This XDR routine returns a structure, which contains
   the result of the user's request, to the user's calling process. */

bool_t xdr_returntype(xdrsp,ptr)
XDR *xdrsp;
struct result *ptr;
{
  if (!xdr_int(xdrsp,&ptr->childproc_id))
    return(FALSE);
  if (!xdr_int(xdrsp,&ptr->status))
    return(FALSE);
  if (!xdr_union(xdrsp,&ptr->utype,&ptr->uval,u_tag_arms,NULL))
    return(FALSE);
  return(TRUE);
}



/* This XDR routine (de)serializes all the
   information about the user's request.     */

bool_t xdr_request(xdrsp,ptr)
XDR *xdrsp;
request *ptr;
{int i;
 int maxapp=req_applength+2;
 int maxcmd=req_cmdlength+2;
 int maxdb=req_dblength+2;
 int maxentry=req_entrylength+2;
 int maxname=namelen+2;
 int onebyte=1;
    for (i=0; i<maxapp; i++)
    { if (!xdr_bytes(xdrsp,ptr->application[i],&onebyte,onebyte))
        return(FALSE);    }
    for (i=0; i<maxcmd; i++)
     { if (!xdr_bytes(xdrsp,ptr->command[i],&onebyte,onebyte))
         return(FALSE);   }
    for (i=0; i<maxdb; i++)
```

```
     {  if (!xdr_bytes(xdrsp,ptr->dbname[i],&onebyte,onebyte))
          return(FALSE);    }
    for (i=0; i<maxentry; i++)
     { if (!xdr_bytes(xdrsp,ptr->entry[i],&onebyte,onebyte))
         return(FALSE);    }
    for (i=0; i<maxname; i++)                                  /* vary with the */
     { if (!xdr_bytes(xdrsp,ptr->student[i],&onebyte,onebyte)) /*nature of the*/
         return(FALSE);    }                                    /* application.*/
    return(TRUE);
}


/* This XDR routine is constructed particularly for
   the access manager to accepts ID information from
   caller processes.                                  */

bool_t xdr_idinfo(xdrsp,ptr)
XDR *xdrsp;
id_info *ptr;
{int i, maxsize=maxhostname;
 int onebyte=1;
   for (i=0; i<maxsize; i++)
     { if (!xdr_bytes(xdrsp,ptr->hostname[i],&onebyte,onebyte))
         return(FALSE);   }
   if (!xdr_int(xdrsp,&ptr->id))
     return(FALSE);
   return(TRUE);
}


/* This is the main body of this module which comprises two
   infinite processes.  The first process, a child process of
   this module created by a fork system call, determines the
   next user of the DCSUNIX system whereas the second one,
   the parent process, is merely used for the registration of
   the two RPCs defined previously at the local host "bylands".  */

main()
{ int proc_id;
  char caller_hostname[maxhostname];
  char command[cmdlength];
  char proc_str[maxdigits];

     pipe(host_fildes);  /* sets up a FIFO queue using two Unix pipes  */
     pipe(procnb_fildes);
     pid=fork();          /* the process number of the child process */
     if (pid==0)          /* will be returned to the parent process  */
     {
        while(1==1)
        {
           read(host_fildes[0],caller_hostname, maxhostname);
           read(procnb_fildes[0],&proc_id,sizeof(int));
           restart_proc(caller_hostname,proc_id);
           pid=getpid();  /* the process number of the child process itself. */
           strcpy(command,"kill -17 ");
           itos(proc_str,pid);
```

```
          strcat(command,proc_str);
          system(command);
        }
    }
    else
    {
      registerrpc(0x20000001,1,1,queue_manager,xdr_idinfo,xdr_void);
      registerrpc(0x20000001,1,2,access_manager,xdr_request,xdr_returntype);
      svc_run();
    }
}


/*  For the second experiment, the following modifications are
    required while keeping the other parts fixed.

    (1) change the structure request to:

            typedef struct result
            {
               char application[req_applength+2];
               char command[req_cmdlength+2];
               char dbname[req_dblength+2];
               char entry[req_entrylength+2];
               student_record data;
            } result;

     (2) delete the enumeration type uniontype;

     (3) change the structure result to:

             struct result
             {
                int childproc_id;
                bool_t  status;
                char *errormsg;
             };

     (4) For the second RPC access_manager:

         (a) delete the statement

                  fprintf(fp,"%s\n",request_info->student);

         (b) add the following statements just after the first fclose statement

                fp=fopen(datafile,"w");
                fprintf(fp,"%s\n",request_info->data.name);
                fprintf(fp,"%d\n",request_info->data.age);
                fprintf(fp,"%c\n",request_info->data.sex);
                fprintf(fp,"%d\n",request_info->data.house_nb);
                fprintf(fp,"%s\n",request_info->data.street);
                fprintf(fp,"%s\n",request_info->data.town);
                fprintf(fp,"%s\n",request_info->data.department);
                fprintf(fp,"%d\n",request_info->data.year);

         (c) delete the if statement

     (5) delete routines xdr_error,xdr_valueand the array of
         structures u_tag_arms

     (6) For routine xdr_returntype, change the last if statement as:

            if (!xdr_string(xdrsp,&ptr->errormsg,255))
              return(FALSE);

     (7) For the routine xdr_request

         (a) adds the following variable initializations

                 int namelg=namelen;
                 int streetlg=streetlen;
                 int townlg=townlen;
                 int departlg=departlen;

         (b) replace the last for statement by the following statements:

                for (i=0; i<namelg; i++)
                {  if (!xdr_bytes(xdrsp,ptr->name[i],&onebyte,onebyte))
                     return(FALSE);  }
                if (!xdr_int(xdrsp,&ptr->age))
                  return(FALSE);
                if (!xdr_bytes(xdrsp,&ptr->sex,&onebyte,onebyte))
                  return(FALSE);
                if (!xdr_int(xdrsp,&ptr->house_nb))
                  return(FALSE);
                for (i=0; i<streetlg; i++)
                {  if (!xdr_bytes(xdrsp,ptr->street[i],&onebyte,onebyte))
                     return(FALSE);  }
                for (i=0; i<townlg; i++)
                {  if (!xdr_bytes(xdrsp,ptr->town[i],&onebyte,onebyte))
                     return(FALSE);  }
                for (i=0; i<departlg; i++)
                {  if (!xdr_bytes(xdrsp,ptr->department,&onebyte,onebyte))
                     return(FALSE);   }
                if (!xdr_int(xdrsp,&ptr->year))
                  return(FALSE);
                return(TRUE);


As this example illustrated, whenever a new experiment is required,
the structures request and result together with their associated
XDR (de)serialization routines must be changed to meet the requirement.
As the last example of the structured approach, the following changes
must be made to this module in order to carry out the sixth experiment
mentioned in the main thesis - attempts to access a particular field, say
the department field, of a student record stored in the data bank.

(1) change the structure request as:

       typedef struct request
```

```
{
   char application[req_applength+2];
   char command[req_cmdlength+2];
   char dbname[req_dblength+2];
   char entry[req_entrylength+2];
   char student[namelen+2];
   char departfield[departlen+2];
} request;
```

(2) the structure result will be defined as follows:

```
struct result
{ enum uniontype   utype;

    int childproc_id;
    bool_t  status;
    union
    {
      char value[maxlength];
      char *errormsg;
      char *dont;
    } uval;
};
```

(3) For the second RPC access_manager :

(a) the following extra statement is added just before the first fclose statement:

```
fprintf(fp,"%s\n",request->info.departfield);
```

(b) the first if statement is substituted by another if statement:

```
if (answer->status==TRUE)
{  answer->utype=val;
   fscanf(fp,"%s\n",answer->uval.value);    }
else
{  answer->utype=error;
   fscanf(fp,"%s",answer->uval.errormsg); }
```

(4) The routine xdr_value is modified as:

```
bool_t xdr_value(xdrsp,ptr)
XDR *xdrsp;
char *ptr;
{int i;
 int onebyte=1;
 int maxsize=departlen;
    for (i=0; i<maxsize; i++)
    {  if (!xdr_bytes(xdrsp,ptr[i],&onebyte,onebyte))
         return(FALSE);    }
    return(TRUE);
}
```

(5) Finally, the routine xdr_returnedtype must be changed accordingly as:

```
bool_t xdr_returntype(xdrsp,ptr)
XDR *xdrsp;
struct result *ptr;
{
   if (!xdr_int(xdrsp,&ptr->childproc_id))
     return(FALSE);
   if (!xdr_int(xdrsp,&ptr->status))
     return(FALSE);
   if (!xdr_union(xdrsp,&ptr->utype,&ptr->uval,u_tag_arms,NULL))
     return(FALSE);
   return(TRUE);
}
```

```c
/* This module is constructed for the first stage testing, that is all
   the experiments mentioned in section 7.2.1.3. of the main thesis, of
   the stack example using the un-structured remote data access method.
   The key difference between this approach and the previous approach
   is the way of handling data.  In this approach, the data part of
   the user's request is treated as rows of bytes with no structure.  */


#include <stdio.h>
#include <string.h>
#include <rpc/rpc.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netdb.h>
#include <netinet/in.h>

#define server_machine "bylands"
#define client_machine "ws_lodge1"


#define maxhostname 20
#define pathlength 80
#define maxlength 20
#define maxdigits 11
#define cmdlength 100
#define errorlength 255
#define req_applength 20
#define req_cmdlength 10
#define req_dblength 20
#define req_entrylength 30
#define req_datalength 20
#define maxbuffer 1024        /* the maximum buffer size available
                                 for a single RPC currently.        */


/* define boolean variables */
#define bool_t int
#define TRUE 1
#define FALSE 0



/*  Identification details of the client process */

typedef struct id_info
{
    char hostname[maxhostname];
    int id;
} id_info;


/* This definition may vary according to the user's intention
   ; however, no string is allowed because all data structures
   are converted into an array of characters internally by DCSUNIX.  */

typedef struct user_structure
{
  char datavalues[req_datalength+2];
}  user_structure;


typedef struct request
{
   char application[req_applength+2]; /* the extra two bytes are required in */
   char command[req_cmdlength+2];      /* to satisfy the syntax of a string in*/
   char dbname[req_dblength+2];        /* the language PS-algol.              */
   char entry[req_entrylength+2];
   char data[maxbuffer];        /* this definition does not need any change*/
} request;                      /* even when the nature of the application */
                                /* has changed because they all treated   */
                                /* as rows of bytes.                      */


/* the structure below is used to store the result
        after the user's request is committed      */

enum uniontype {  val=1, error=2, dontcare=3 } ;
   /* enumeration type specifications of the union  */
   /* inside the structure result below.*/

/* A typedef construct can also be used for the following C structure,
   however no matter which way is used, the current compiler always
   gives an enumeration type clash warning despite that the program
   can still run successfully.                                   */

struct result
{ enum uniontype   utype;    /* the union's discriminant which aims to
                                    select the arms of the union.     */
    int childproc_id;
    bool_t  status;         /*  1 means success and 0 means fail  */
    union
    {
       char value[maxbuffer];  /* this definition does not any change for
                                  the same reason given for the data field
                                  of the C structure request above.     */

       char *errormsg; /* assumed to have a maximum of 255 characters */
       char *dont; /* this is just an end-marker of the union */
    } uval;
} ;


/* This routine takes in an empty string and an integer, and
   then it converts the integer into the corresponding ASCII string  */

itos(str,intnb)
char str[maxdigits];
int intnb;
```

```
{ int tmp,quot,rem,pos;
  char s[maxdigits],*value;
  tmp= -1;
  do
  {
    ++tmp;
    quot=intnb/10;
    rem=intnb%10;
    switch (rem)
    {
       case 0: s[tmp]='0';
               break;
       case 1: s[tmp]='1';
               break;
       case 2: s[tmp]='2';
               break;
       case 3: s[tmp]='3';
               break;
       case 4: s[tmp]='4';
               break;
       case 5: s[tmp]='5';
               break;
       case 6: s[tmp]='6';
               break;
       case 7: s[tmp]='7';
               break;
       case 8: s[tmp]='8';
               break;
       case 9: s[tmp]='9';
    }
    intnb=quot;
  } while (quot!=0);
  for (pos=0; tmp!= -1; ++pos)
  {
      value= &s[tmp--];
      str[pos]= *value;
  }
}

/* The following three XDR routines and a C structure are required
   for the (de)serialization of the union part of the structure
   that is going to be returned to this.                        */

bool_t xdr_error(xdrsp,ptr)
XDR *xdrsp;
char *ptr;
{
   if (!xdr_string(xdrsp,&ptr,255))
     return(FALSE);
   return(TRUE);
}


bool_t xdr_onebyte(xdrsp,ptr)
XDR *xdrsp;
char *ptr;
{ int onebyte=1;
     if (!xdr_bytes(xdrsp,&ptr,&onebyte,onebyte))
       return(FALSE);
     return(TRUE);
}


bool_t xdr_value(xdrsp,ptr)
XDR *xdrsp;
char *ptr;
{int onebyte=1;
 int size=maxbuffer;
    if (!xdr_array(xdrsp,&ptr,&size,size,onebyte,xdr_onebyte))
         return(FALSE);
    return(TRUE);
}

struct xdr_discrim u_tag_arms[3] =     /* the definition of structure    */
{                                      /* xdr_discrim is located in      */
   { val, xdr_value},                  /* the header file <rpc/rpc.h>    */
   { error, xdr_error},
   { dontcare , NULL}
};


/* This XDR routine returns a structure, which contains
   the result of the user's request, to the user's calling process. */

bool_t xdr_returntype(xdrsp,ptr)
XDR *xdrsp;
struct result *ptr;
{
   if (!xdr_int(xdrsp,&ptr->childproc_id))
     return(FALSE);
   if (!xdr_int(xdrsp,&ptr->status))
     return(FALSE);
   if (!xdr_union(xdrsp,&ptr->utype,&ptr->uval,u_tag_arms,NULL))
     return(FALSE);
   return(TRUE);
}



/* This XDR routine (de)serializes all the
   information about the user's request.     */

bool_t xdr_request(xdrsp,ptr)
XDR *xdrsp;
request *ptr;
{int i;
 int maxapp=req_applength+2;
 int maxcmd=req_cmdlength+2;
 int maxdb=req_dblength+2;
```

```
 int maxentry=req_entrylength+2;
 int size=maxbuffer;
 int onebyte=1;
    for (i=0; i<maxapp; i++)
    { if (!xdr_bytes(xdrsp,ptr->application[i],&onebyte,onebyte))
        return(FALSE);   }
    for (i=0; i<maxcmd; i++)
    { if (!xdr_bytes(xdrsp,ptr->command[i],&onebyte,onebyte))
        return(FALSE);  }
    for (i=0; i<maxdb; i++)
    {  if (!xdr_bytes(xdrsp,ptr->dbname[i],&onebyte,onebyte))
         return(FALSE);   }
    for (i=0; i<maxentry; i++)
    { if (!xdr_bytes(xdrsp,ptr->entry[i],&onebyte,onebyte))
        return(FALSE);   }
    if (!xdr_array(xdrsp,ptr->data,&size,size,     /* this would be suitable*/
                  onebyte,xdr_onebyte))            /* for all types of stacks*/
      return(FALSE);
    return(TRUE);
}


/* This XDR routine is constructed particularly for
   the access manager to accepts ID information from
   caller processes.                                */

bool_t xdr_idinfo(xdrsp,ptr)
XDR *xdrsp;
id_info *ptr;
{int i, maxsize=maxhostname;
 int onebyte=1;
    for (i=0; i<maxsize; i++)
    { if (!xdr_bytes(xdrsp,ptr->hostname[i],&onebyte,onebyte))
        return(FALSE);  }
    if (!xdr_int(xdrsp,&ptr->id))
      return(FALSE);
    return(TRUE);
}


/* This routine invokes the queue_manager of the "system co-ordinator"
   at the server site so as to register this client process on the
   server's FIFO queue.                                          */

bool_t callrpc_queue_manager(process_ID)
id_info process_ID;
{
   struct hostent *hp;       /* structure hostent defined in <netdb.h> */
   struct timeval pertry_timeout,total_timeout;  /* structure timeval defined
                                                   in <time.h>    */
   struct sockaddr_in server_addr;    /* structure sockaddr_in defined
                                          in <netinet/in.h>        */
   int sock= RPC_ANYSOCK;             /*  constant RPC_ANYSOCK defined in
                                          <rpc/svc.h> which is automatically
                                          included by the file <rpc/rpc.h>.
                                          The purpose of this variable is to
                                          ask the kernel to choose the most
                                          appropriate socket for establishing
                                          the rpc connection.         */
   register CLIENT *client;

      if ((hp=gethostbyname(server_machine))==NULL)
      {
        printf("Can't get address for remote machine %s\n",server_machine);
        return(FALSE);
      }
      pertry_timeout.tv_sec=1;      /*  timeout interval for each rpc    */
      pertry_timeout.tv_usec=0;     /*  call in seconds and microseconds */
      bcopy(hp->h_addr,(struct in_addr *)&server_addr.sin_addr,
            hp->h_length);               /* construct Internet address of */
      server_addr.sin_family=AF_INET;   /* the server by putting the name*/
      server_addr.sin_port=0;           /* family address and port number*/
                                        /* of the host. If the port number*/
                                        /* is 0, then the remote portmapper*/
                                        /* will be consulted to get the  */
                                        /* actual port of the remote     */
                                        /* program. Constant AF_INET is  */
                                        /* defined in <sys/socket.h> and */
                                        /* structure in_addr defined in  */
                                        /* <netinet/in.h>                */
      if ((client=clntudp_create(&server_addr,0x20000001,1, /*create a rpc*/
                              pertry_timeout,&sock))==NULL)   /*client handle*/
      {                                                       /* for the remote*/
          printf("Can't create RPC handle\n");                /* program.      */
          return(FALSE);
      }
      total_timeout.tv_sec=1;    /*  the total timeout interval of the     */
      total_timeout.tv_usec=0;   /*  rpc in seconds and microseconds which */
                                 /*  has the same value as the timeout of */
                                 /*  each try and therefore only one call */
                                 /*  is issued each time.                 */
      if (clnt_call(client,1,xdr_idinfo,&process_ID,    /* this is a micro */
                xdr_void,0,total_timeout)!=RPC_SUCCESS)/* which calls the */
      {                                                  /* remote procedure*/
         printf("The remote server is either");          /* associated with */
         printf(" busy or down.  Please try again ");   /* the client      */
         printf("later.\n");                             /* handle created  */
         return(FALSE);                                  /* above.          */
      }

      clnt_destroy(client);      /* deallocate the space assocated */
                                 /* with the client handle        */
      return(TRUE);
}


/* This routine invokes the access_manager of the "system co-ordinator"
   at the server site so as to access the required abstract object stored
   in the PS-algol database system.                               */
```

```
struct result *callrpc_access_manager(example)
request example;
{
   struct hostent *hp;       /* structure hostent defined in <netdb.h> */
   struct timeval pertry_timeout,total_timeout;  /* structure timeval defined
                                              in <time.h>              */
   struct sockaddr_in server_addr;    /* structure sockaddr_in defined
                                          in <netinet/in.h>          */
   int sock= RPC_ANYSOCK;        /*  constant RPC_ANYSOCK defined in
                                     <rpc/svc.h> which is automatically
                                     included by the header file <rpc/rpc.h>.
                                     the purpose of this variable is to ask
                                     the kernel to choose the appropriate
                                     socket to establish the rpc connection.*/

   struct result *reply;

   char command[cmdlength];
   char proc_str[maxdigits];

   register CLIENT *client;

       if ((hp=gethostbyname(server_machine))==NULL)
       {
          printf("Can't get address for remote machine %s\n",server_machine);
          printf("Please try again later.\n");
          exit(0);
       }
       pertry_timeout.tv_sec=6;       /*  timeout interval for each rpc    */
       pertry_timeout.tv_usec=0;      /*  call in seconds and microseconds */
       bcopy(hp->h_addr,(struct in_addr *)&server_addr.sin_addr,
             hp->h_length);             /* construct Internet address of */
       server_addr.sin_family=AF_INET; /* the server by putting the name*/
       server_addr.sin_port=0;          /* family address and port number*/
                                        /* of the host. If port number  */
                                        /* is 0, then the remote portmapper*/
                                        /* will be consulted to get the  */
                                        /* actual port of the remote     */
                                        /* program. Constant AF_INET is  */
                                        /* defined in <sys/socket.h> and */
                                        /* structure in_addr defined in  */
                                        /* <netinet/in.h>                */
       if ((client=clntudp_create(&server_addr,0x20000001,1, /*create a rpc */
                              pertry_timeout,&sock))==NULL)   /*client handle*/
       {                                                   /* for the remote*/
          printf("Can't create RPC handle\n");             /* program.      */
          printf("Please try again later.\n");
          exit(0);
       }
       total_timeout.tv_sec=30;   /*  the total timeout interval of the     */
       total_timeout.tv_usec=0;  /*  rpc call in seconds and microseconds. */
                                 /*  Since this timeout value is 5 times   */
                                 /*  larger than the per try timeout value */
                                 /*  so there may have a maximum of 5 tries.*/
       reply=(struct result *)malloc(sizeof(struct result));
```

```
       if ((clnt_call(client,2,xdr_request,&example,            /* this is */
            xdr_returntype,reply,total_timeout))!=RPC_SUCCESS)  /* a micro */
       {                                                    /* which calls the */
          printf("The remote server is either");           /* remote procdure */
          printf(" busy or down.  Please try again ");     /* associated with */
          printf("later.\n");                              /* the client handle*/
          exit(0);                                         /* created above.  */
       }
       clnt_destroy(client);       /* deallocate the space assocated */
                                   /* the client handle.        */
       strcpy(command,"rsh bylands kill -19 "); /* restart the child process*/
       itos(proc_str,reply->childproc_id);      /* of the server's "system  */
       strcat(command,proc_str);                /* co-ordinator".           */
       system(command);
       return (reply);
}


/* The main program body for the first experiment mentioned
   in section 7.2.1.3, which is the read mode, using the
   un-structured approach.                                       */

main()
{request example1;
 user_structure in_info,*out_info;   /* in_info is needed for the write mode */
 struct result *reply;              /* while out_info is required for the   */
 id_info process_ID;                /* read mode.                           */
 char command[cmdlength];
 char proc_str[maxdigits];
 char *row_of_bytes;
   strcpy(example1.application,"\"StringStack\"");
   strcpy(example1.command,"\"top\"");
   strcpy(example1.dbname,"\"utility\"");
   strcpy(example1.entry,"\"application1\"");
   strcpy(process_ID.hostname,server_machine);
   process_ID.id=getpid();
   if (callrpc_queue_manager(process_ID)==TRUE)
   {
     strcpy(command,"kill -17 ");   /* signal a sleep signal to itself*/
     itos(proc_str,process_ID.id);  /* and then waiting for service.  */
     strcat(command,proc_str);
     system(command);
     reply=(struct result *)malloc(sizeof(struct result));  /* this is the   */
     reply=callrpc_access_manager(example1);                /* re-start point*/
     if (reply->status==TRUE)
     {
        out_info=(user_structure *)malloc(sizeof(user_structure));
        out_info=(user_structure *)reply->uval.value;     /* this converts */
        printf("The top value of the stack is %s\n",      /* the returned  */
                out_info->datavalues);                     /* row of bytes  */
     }                                                     /* back to the   */
     else                                                  /* user's defined */
     {                                                     /* structure.    */
        printf("%s\n",reply->uval.errormsg);
     }
```

```
   }
}


/* The main program body for the second experiment
   using the un-structured approach. This time the
   experiment is in the write mode.

main()
{request example2;
 user_structure in_info,*out_info;
 struct result *reply;
 id_info process_ID;
 char command[cmdlength];
 char proc_str[maxdigits];
 char *row_of_bytes;
     strcpy(example2.application,"\"StringStack\"");
     strcpy(example2.command,"\"push\"");
     strcpy(example2.dbname,"\"utility\"");
     strcpy(example2.entry,"\"application1\"");
     strcpy(in_info.datavalues,"\"hello\"");
     row_of_bytes=(char *)malloc(sizeof(user_structure));
     row_of_bytes=(char*)&in_info;              /* treat data just as a row  */
     strcpy(example2.data,row_of_bytes);        /* of bytes by means of its  */
     strcpy(process_ID.hostname,server_machine); /* address and size.        */
     process_ID.id=getpid();
     if (callrpc_queue_manager(process_ID)==TRUE)
     {
        strcpy(command,"kill -17 ");
        itos(proc_str,process_ID.id);
        strcat(command,proc_str);
        system(command);
        reply=(struct result *)malloc(sizeof(struct result));
        reply=callrpc_access_manager(example2);
        if (reply->status==TRUE)
        {
           printf("Successfully push the string\n");
        }
        else
        {
             printf("%s\n",reply->uval.errormsg);
        }
     }
}             */


/*  As the two examples illustrated above, whenever data is to be
    transmitted to the server, it will be converted into a row of
    characters; whereas in case of data is being received from the
    server, it is converted back to the user-defined structure
    before performing any interpretation on it.  The mian program
    body for the fourth experiment mentioned in section 7.2.1.3.
    is constructed in a combination of the two technique described above.  */
```

```
/* This module is designated as the "system co-ordinator" of the
   DCSUNIX system for the first stage testing of the stack(and the
   student databank) example using the un-structured remote data
   access method. The main difference between this module and the
   one for the structured approach is that the former can be used
   for types of stack,independent of the nature of the data items
   placed on the stack.   */


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

/* define limits */

#define maxhostname 20
#define pathlength 80
#define maxlength 20
#define maxdigits 11
#define cmdlength 100
#define errorlength 255
#define req_applength 20
#define req_cmdlength 10
#define req_dblength 20
#define req_entrylength 30
#define maxbuffer 1024       /* the maximum buffer size allowed
                                for a single RPC currently.     */

/* define boolean variables */
#define bool_t int
#define TRUE 1
#define FALSE 0

/* define locations of the two Unix files and the PS-algol program "service" */

#define cmdfile "/usr/lib/ps/dbs/cmdfile"
#define datafile "/usr/lib/ps/dbs/datafile"
#define service "/user/res/mscwct/psalgol/service.out"


/*  implementation of the server's FIFO queue */
int host_fildes[2];
int procnb_fildes[2];

int pid; /* refers to the system co-ordinator's
            child process ID number               */



/*  Identification details of the client process */

typedef struct id_info
{
   char hostname[maxhostname];
   int id;
} id_info;
```

```
/*   the overall format of the user's request  */

typedef struct request
{
   char application[req_applength+2]; /* the extra two bytes are required in */
   char command[req_cmdlength+2];      /* to satisfy the syntax of a string  */
   char dbname[req_dblength+2];        /* in the langauge PS-algol.          */
   char entry[req_entrylength+2];
   char data[maxbuffer];  /* This definition does not need any change even  */
} request;                /* the nature of the application has changed      */
                          /* because all data structures are simply treated */
                          /* as rows of bytes.                             */


/* the structure below is used to store the result
        after the user's request is committed      */

enum uniontype {  val=1, error=2, dontcare=3 } ;
   /* enumeration type specifications of the union  */
   /* inside the structure result below.*/

/* A typedef construct can also be used for the following C structure,
   however no matter which way is used, the current compiler always
   gives an enumeration type clash warning despite that the program
   can still run successfully.                               */

struct result
{ enum uniontype   utype;    /* the union's discriminant which aims to
                                  select the arms of the union.      */
    int childproc_id;
    bool_t  status;         /*  1 means success and 0 means fail  */
    union
    {
       char value[maxbuffer]; /* this definition is fixed for the same reason
                                 explained about the data field of structure
                                 request above.              */

       char *errormsg; /* assumed to have a maximum of 255 characters */
       char *dont; /* this is just an end-marker of the union */
    } uval;
} ;


/* This routine takes in an empty string and an integer, and then
   it converts the integer into the corresponding ASCII string  */

itos(str,intnb)
char str[maxdigits];
int intnb;
{ int tmp,quot,rem,pos;
  char s[maxdigits],*value;
  tmp= -1;
  do
  {
    ++tmp;
    quot=intnb/10;
    rem=intnb%10;
    switch (rem)
    {
       case 0: s[tmp]='0';
               break;
       case 1: s[tmp]='1';
               break;
       case 2: s[tmp]='2';
               break;
       case 3: s[tmp]='3';
               break;
       case 4: s[tmp]='4';
               break;
       case 5: s[tmp]='5';
               break;
       case 6: s[tmp]='6';
               break;
       case 7: s[tmp]='7';
               break;
       case 8: s[tmp]='8';
               break;
       case 9: s[tmp]='9';
    }
    intnb=quot;
  } while (quot!=0);
  for (pos=0; tmp!= -1; ++pos)
  {
      value= &s[tmp--];
      str[pos]= *value;
  }
}


/* This routine takes in the name of the machine of a client
   process, together with the ID number of the process in
   that machine, then it will send a signal to re-start that process.*/

restart_proc(hostname,proc_id)
char hostname[maxhostname];
int proc_id;
{ char command[cmdlength];
  char proc_str[maxdigits];
     strcpy(command,"rsh ");
     strcat(command,hostname);
     strcat(command," kill -19 ");
     itos(proc_str,pid);
     strcat(command,proc_str);
     system(command);
}
```

```c
/* This manager puts all the caller processes's
   ID information into the server's FIFO queue.
   This routine will be registered as a RPC shortly */

queue_manager(client_info)
id_info *client_info;
{
   write(host_fildes[1],client_info->hostname,maxhostname);
   write(procnb_fildes[1],&client_info->id,sizeof(int));
}


/* This is a second RPC which sends all the information
   about a user's request to the two Unix files cmdfile
   and datafile accordingly, then those information will
   be consumed by the PS-algol "service" to carry out the
   user's request.                                    */

struct result *access_manager(request_info)
request *request_info;
{FILE *fp;
 char executeprog[pathlength];
 struct result *answer;

   fp=fopen(cmdfile,"w");                            /* transfer information to */
   fprintf(fp,"%s\n",request_info->application);    /* the file cmdfile in     */
   fprintf(fp,"%s\n",request_info->command);        /* order to execute the    */
   fprintf(fp,"%s\n",request_info->dbname);         /* client's request        */
   fprintf(fp,"%s\n",request_info->entry);
   fclose(fp);

   fp=fopen(datafile,"w");                          /* this part will have no  */
   fprintf(fp,"%s\n",request_info->data);           /* effect on file datafile */
   fclose(fp);                                      /* if the user's request  */
                                                    /* is in the read mode. */
   strcpy(executeprog,"psr");
   strcat(executeprog,service);
   system(executeprog);         /* carry out the user's request.*/

   answer=(struct result*)malloc(sizeof(struct result));
   answer->childproc_id=pid;    /* this part retruns the result of the request*/
   fp=fopen(datafile,"r");
   fscanf(fp,"%d\n",&answer->status);
   if (answer->status==TRUE)
   {  answer->utype=val;
      fscanf(fp,"%s\n",answer->uval.value);   }
   else
   {   answer->utype=error;
       fscanf(fp,"%s",answer->uval.errormsg); }
   fclose(fp);
   return(answer);
}


/* The following three XDR routines and a C structure are required
   for the (de)serialization of the union part of the structure
   that is going to be returned to the user's calling process. */

bool_t xdr_error(xdrsp,ptr)
XDR *xdrsp;
char *ptr;
{
   if (!xdr_string(xdrsp,&ptr,255))
     return(FALSE);
   return(TRUE);
}


bool_t xdr_onebyte(xdrsp,ptr)
XDR *xdrsp;
char *ptr;
{  int onebyte=1;
      if (!xdr_bytes(xdrsp,&ptr,&onebyte,onebyte))
        return(FALSE);
      return(TRUE);
}

bool_t xdr_value(xdrsp,ptr)
XDR *xdrsp;
char *ptr;
{int onebyte=1;
 int size=maxbuffer;
    if (!xdr_array(xdrsp,&ptr,&size,size,onebyte,xdr_onebyte))
         return(FALSE);
    return(TRUE);
}

struct xdr_discrim u_tag_arms[3] =    /* the definition of structure    */
{                                     /* xdr_discrim is located in      */
   { val, xdr_value},                 /* the header file <rpc/rpc.h>    */
   { error, xdr_error},
   { dontcare , NULL}
};


/* This XDR routine returns a structure, which contains
   the result of the user's request, to the user's calling process. */

bool_t xdr_returntype(xdrsp,ptr)
XDR *xdrsp;
struct result *ptr;
{
  if (!xdr_int(xdrsp,&ptr->childproc_id))
    return(FALSE);
  if (!xdr_int(xdrsp,&ptr->status))
    return(FALSE);
  if (!xdr_union(xdrsp,&ptr->utype,&ptr->uval,u_tag_arms,NULL))
    return(FALSE);
  return(TRUE);
}
```

```
/* This XDR routine (de)serializes all the
   information about the user's request.     */

bool_t xdr_request(xdrsp,ptr)
XDR *xdrsp;
request *ptr;
{int i;
 int maxapp=req_applength+2;
 int maxcmd=req_cmdlength+2;
 int maxdb=req_dblength+2;
 int maxentry=req_entrylength+2;
 int onebyte=1;
 int size=maxbuffer;
    for (i=0; i<maxapp; i++)
    { if (!xdr_bytes(xdrsp,ptr->application[i],&onebyte,onebyte))
        return(FALSE);    }
    for (i=0; i<maxcmd; i++)
     { if (!xdr_bytes(xdrsp,ptr->command[i],&onebyte,onebyte))
         return(FALSE);   }
    for (i=0; i<maxdb; i++)
     {  if (!xdr_bytes(xdrsp,ptr->dbname[i],&onebyte,onebyte))
          return(FALSE);    }
    for (i=0; i<maxentry; i++)
     { if (!xdr_bytes(xdrsp,ptr->entry[i],&onebyte,onebyte))
         return(FALSE);    }
    if (!xdr_array(xdrsp,ptr->data,&size,size,      /* this would be suitable*/
                   onebyte,xdr_onebyte))             /* for all types of stacks*/
      return(FALSE);
    return(TRUE);
}


/* This XDR routine is constructed particularly for
   the access manager to accepts ID information from
   caller processes.                                  */

bool_t xdr_idinfo(xdrsp,ptr)
XDR *xdrsp;
id_info *ptr;
{int i, maxsize=maxhostname;
 int onebyte=1;
   for (i=0; i<maxsize; i++)
     { if (!xdr_bytes(xdrsp,ptr->hostname[i],&onebyte,onebyte))
         return(FALSE);   }
   if (!xdr_int(xdrsp,&ptr->id))
     return(FALSE);
   return(TRUE);
}


/* This is the main body of this module which comprises two
   infinite processes.  The first process, a child process of
   this module created by a fork system call, determines the
   next user of the DCSUNIX system whereas the second one,
   the parent process, is merely used for the registration of
   the two RPCs defined previously at the local host "bylands".  */

main()
{ int proc_id;
  char caller_hostname[maxhostname];
  char command[cmdlength];
  char proc_str[maxdigits];

     pipe(host_fildes);  /* sets up a FIFO queue using two Unix pipes  */
     pipe(procnb_fildes);
     pid=fork();         /* the process number of the child process */
     if (pid==0)         /* will be returned to the parent process  */
     {
        while(1==1)
        {
           read(host_fildes[0],caller_hostname, maxhostname);
           read(procnb_fildes[0],&proc_id,sizeof(int));
           restart_proc(caller_hostname,proc_id);
           pid=getpid();  /* the process number of the child process itself. */
           strcpy(command,"kill -17 ");
           itos(proc_str,pid);
           strcat(command,proc_str);
           system(command);
        }
     }
     else
     {
       registerrpc(0x20000001,1,1,queue_manager,xdr_idinfo,xdr_void);
       registerrpc(0x20000001,1,2,access_manager,xdr_request,xdr_returntype);
       svc_run();
     }
}
```

```c
/* This module is constructed for the first stage testing, that is all
   the experiments mentioned in section 7.2.2.3. of the main thesis except
   experiments (e) and (f), of the student databank example using the
   un-structured remote data access method. The key difference between
   this approach and the previous approach is the way of handling data.In this
   approach, the data part of the user's request is treated as rows of
   bytes with no structure.  In order to be consistent with the example
   given for the structured approach (module s2_client_process.c), the
   same operation is performed using this approach. */



#include <stdio.h>
#include <string.h>
#include <rpc/rpc.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netdb.h>
#include <netinet/in.h>

#define server_machine "bylands"
#define client_machine "ws_lodgel"


#define maxhostname 20
#define pathlength 80
#define maxlength 20
#define maxdigits 11
#define cmdlength 100
#define errorlength 255
#define req_applength 20
#define req_cmdlength 10
#define req_dblength 20
#define req_entrylength 30
#define namelen 30
#define streetlen 30
#define townlen 20
#define departlen 30
#define maxbuffer 1024       /* the maximum buffer size available
                                for a single RPC currently.      */


/* define boolean variables */
#define bool_t int
#define TRUE 1
#define FALSE 0



/*  Identification details of the client process */

typedef struct id_info
{
    char hostname[maxhostname];
    int id;
} id_info;


typedef struct student_record
{
  char name[namelen];
  int age;
  char sex;
  int house_nb;
  char street[streetlen];
  char town[townlen];
  char department[departlen];
  int year;
}  student_record;


typedef struct request
{
   char application[req_applength+2]; /* the extra two bytes are required in */
   char command[req_cmdlength+2];     /* to satisfy the syntax of a string in*/
   char dbname[req_dblength+2];       /* the language PS-algol.              */
   char entry[req_entrylength+2];
   char student[maxlength+2];
} request;



/* the structure below is used to store the result
        after the user's request is committed      */

enum uniontype {  val=1, error=2, dontcare=3 } ;
   /* enumeration type specifications of the union  */
   /* inside the structure result below.*/

/* A typedef construct can also be used for the following C structure,
   however no matter which way is used, the current compiler always
   gives an enumeration type clash warning despite that the program
   can still run successfully.                                    */

struct result
{ enum uniontype   utype;    /* the union's discriminant which aims to
                                  select the arms of the union.      */
    int childproc_id;
    bool_t  status;         /*  1 means success and 0 means fail  */
    union
    {
       char value[maxbuffer];  /* this definition does not any change for
                                  the same reason given for the data field
                                  of the C structure request above.      */

       char *errormsg; /* assumed to have a maximum of 255 characters */
       char *dont; /* this is just an end-marker of the union */
    } uval;
} ;
```

```c
/* This routine takes in an empty string and an integer, and
   then it converts the integer into the corresponding ASCII string  */

itos(str,intnb)
char str[maxdigits];
int intnb;
{ int tmp,quot,rem,pos;
  char s[maxdigits],*value;
  tmp= -1;
  do
  {
    ++tmp;
    quot=intnb/10;
    rem=intnb%10;
    switch (rem)
    {
       case 0: s[tmp]='0';
               break;
       case 1: s[tmp]='1';
               break;
       case 2: s[tmp]='2';
               break;
       case 3: s[tmp]='3';
               break;
       case 4: s[tmp]='4';
               break;
       case 5: s[tmp]='5';
               break;
       case 6: s[tmp]='6';
               break;
       case 7: s[tmp]='7';
               break;
       case 8: s[tmp]='8';
               break;
       case 9: s[tmp]='9';
    }
    intnb=quot;
  } while (quot!=0);
  for (pos=0; tmp!= -1; ++pos)
  {
      value= &s[tmp--];
      str[pos]= *value;
  }
}


/* The following three XDR routines and a C structure are required
   for the (de)serialization of the union part of the structure
   that is going to be returned to this.                     */

bool_t xdr_error(xdrsp,ptr)
XDR *xdrsp;
char *ptr;
{
   if (!xdr_string(xdrsp,&ptr,255))
     return(FALSE);
   return(TRUE);
}


bool_t xdr_onebyte(xdrsp,ptr)
XDR *xdrsp;
char *ptr;
{ int onebyte=1;
     if (!xdr_bytes(xdrsp,&ptr,&onebyte,onebyte))
       return(FALSE);
     return(TRUE);
}


bool_t xdr_value(xdrsp,ptr)
XDR *xdrsp;
char *ptr;
{int onebyte=1;
 int size=maxbuffer;
    if (!xdr_array(xdrsp,&ptr,&size,size,onebyte,xdr_onebyte))
         return(FALSE);
    return(TRUE);
}

struct xdr_discrim u_tag_arms[3] =    /* the definition of structure   */
{                                     /* xdr_discrim is located in     */
   { val, xdr_value},                 /* the header file <rpc/rpc.h>   */
   { error, xdr_error},
   { dontcare , NULL}
};



/* This XDR routine returns a structure, which contains
   the result of the user's request, to the user's calling process. */

bool_t xdr_returntype(xdrsp,ptr)
XDR *xdrsp;
struct result *ptr;
{
   if (!xdr_int(xdrsp,&ptr->childproc_id))
     return(FALSE);
   if (!xdr_int(xdrsp,&ptr->status))
     return(FALSE);
   if (!xdr_union(xdrsp,&ptr->utype,&ptr->uval,u_tag_arms,NULL))
     return(FALSE);
   return(TRUE);
}



/* This XDR routine (de)serializes all the
   information about the user's request.     */
```

```
bool_t xdr_request(xdrsp,ptr)
XDR *xdrsp;
request *ptr;
{int i;
 int maxapp=req_applength+2;
 int maxcmd=req_cmdlength+2;
 int maxdb=req_dblength+2;
 int maxentry=req_entrylength+2;
 int size=maxbuffer;
 int onebyte=1;
    for (i=0; i<maxapp; i++)
    { if (!xdr_bytes(xdrsp,ptr->application[i],&onebyte,onebyte))
        return(FALSE);   }
    for (i=0; i<maxcmd; i++)
    { if (!xdr_bytes(xdrsp,ptr->command[i],&onebyte,onebyte))
        return(FALSE);  }
    for (i=0; i<maxdb; i++)
    {  if (!xdr_bytes(xdrsp,ptr->dbname[i],&onebyte,onebyte))
         return(FALSE);   }
    for (i=0; i<maxentry; i++)
    { if (!xdr_bytes(xdrsp,ptr->entry[i],&onebyte,onebyte))
        return(FALSE);   }
    if (!xdr_array(xdrsp,ptr->data,&size,size,      /* this would be suitable*/
                   onebyte,xdr_onebyte))            /* for all types of stacks*/
      return(FALSE);
    return(TRUE);
}


/* This XDR routine is constructed particularly for
   the access manager to accepts ID information from
   caller processes.                               */

bool_t xdr_idinfo(xdrsp,ptr)
XDR *xdrsp;
id_info *ptr;
{int i, maxsize=maxhostname;
 int onebyte=1;
    for (i=0; i<maxsize; i++)
    { if (!xdr_bytes(xdrsp,ptr->hostname[i],&onebyte,onebyte))
        return(FALSE);  }
    if (!xdr_int(xdrsp,&ptr->id))
      return(FALSE);
    return(TRUE);
}


/* This routine invokes the queue_manager of the "system co-ordinator"
   at the server site so as to register this client process on the
   server's FIFO queue.                                         */

bool_t callrpc_queue_manager(process_ID)
id_info process_ID;
{
   struct hostent *hp;       /* structure hostent defined in <netdb.h> */
   struct timeval pertry_timeout,total_timeout;  /* structure timeval defined
                                                     in <time.h>    */
   struct sockaddr_in server_addr;    /* structure sockaddr_in defined
                                           in <netinet/in.h>         */
   int sock= RPC_ANYSOCK;             /*  constant RPC_ANYSOCK defined in
                                          <rpc/svc.h> which is automatically
                                          included by the file <rpc/rpc.h>.
                                          The purpose of this variable is to
                                          ask the kernel to choose the most
                                          appropriate socket for establishing
                                          the rpc connection.        */
   register CLIENT *client;

         if ((hp=gethostbyname(server_machine))==NULL)
         {
           printf("Can't get address for remote machine %s\n",server_machine);
           return(FALSE);
         }
         pertry_timeout.tv_sec=1;      /*  timeout interval for each rpc    */
         pertry_timeout.tv_usec=0;     /*  call in seconds and microseconds */
         bcopy(hp->h_addr,(struct in_addr *)&server_addr.sin_addr,
               hp->h_length);          /* construct Internet address of */
         server_addr.sin_family=AF_INET;  /* the server by putting the name*/
         server_addr.sin_port=0;       /* family address and port number*/
                                       /* of the host. If the port number*/
                                       /* is 0, then the remote portmapper*/
                                       /* will be consulted to get the  */
                                       /* actual port of the remote     */
                                       /* program. Constant AF_INET is  */
                                       /* defined in <sys/socket.h> and */
                                       /* structure in_addr defined in  */
                                       /* <netinet/in.h>                */
         if ((client=clntudp_create(&server_addr,0x20000001,1, /*create a rpc*/
                       pertry_timeout,&sock))==NULL)   /*client handle*/
         {                                             /* for the remote*/
            printf("Can't create RPC handle\n");       /* program.      */
            return(FALSE);
         }
         total_timeout.tv_sec=1;   /*  the total timeout interval of the     */
         total_timeout.tv_usec=0;  /*  rpc in seconds and microseconds which */
                                   /*  has the same value as the timeout of */
                                   /*  each try and therefore only one call */
                                   /*  is issued each time.                 */
         if (clnt_call(client,1,xdr_idinfo,&process_ID,   /* this is a micro */
                  xdr_void,0,total_timeout)!=RPC_SUCCESS)/* which calls the */
         {                                               /* remote procedure*/
           printf("The remote server is either");        /* associated with */
           printf(" busy or down.  Please try again ");  /* the client      */
           printf("later.\n");                           /* handle created  */
           return(FALSE);                                /* above.          */
         }

         clnt_destroy(client);      /* deallocate the space assocated */
                                    /* with the client handle         */
```

```
        return(TRUE);
}


/* This routine invokes the access_manager of the "system co-ordinator"
   at the server site so as to access the required abstract object stored
   in the PS-algol database system.                             */

struct result *callrpc_access_manager(example)
request example;
{
   struct hostent *hp;       /* structure hostent defined in <netdb.h> */
   struct timeval pertry_timeout,total_timeout;  /* structure timeval defined
                                                     in <time.h>           */
   struct sockaddr_in server_addr;    /* structure sockaddr_in defined
                                           in <netinet/in.h>          */
   int sock= RPC_ANYSOCK;       /*  constant RPC_ANYSOCK defined in
                                    <rpc/svc.h> which is automatically
                                    included by the header file <rpc/rpc.h>.
                                    the purpose of this variable is to ask
                                    the kernel to choose the appropriate
                                    socket to establish the rpc connection.*/

   struct result *reply;

   char command[cmdlength];
   char proc_str[maxdigits];

   register CLIENT *client;

        if ((hp=gethostbyname(server_machine))==NULL)
        {
           printf("Can't get address for remote machine %s\n",server_machine);
           printf("Please try again later.\n");
           exit(0);
        }
        pertry_timeout.tv_sec=6;      /*  timeout interval for each rpc    */
        pertry_timeout.tv_usec=0;     /*  call in seconds and microseconds */
        bcopy(hp->h_addr,(struct in_addr *)&server_addr.sin_addr,
              hp->h_length);            /* construct Internet address of */
        server_addr.sin_family=AF_INET; /* the server by putting the name*/
        server_addr.sin_port=0;         /* family address and port number*/
                                        /* of the host. If port number  */
                                        /* is 0, then the remote portmapper*/
                                        /* will be consulted to get the  */
                                        /* actual port of the remote     */
                                        /* program. Constant AF_INET is  */
                                        /* defined in <sys/socket.h> and */
                                        /* structure in_addr defined in  */
                                        /* <netinet/in.h>                */
        if ((client=clntudp_create(&server_addr,0x20000001,1, /*create a rpc */
                           pertry_timeout,&sock))==NULL)   /*client handle*/
        {                                             /* for the remote*/
           printf("Can't create RPC handle\n");       /* program.      */
           printf("Please try again later.\n");
           exit(0);
        }
        total_timeout.tv_sec=30;   /*  the total timeout interval of the     */
        total_timeout.tv_usec=0;  /*  rpc call in seconds and microseconds. */
                                  /*  Since this timeout value is 5 times   */
                                  /*  larger than the per try timeout value */
                                  /*  so there may have a maximum of 5 tries.*/
        reply=(struct result *)malloc(sizeof(struct result));
        if ((clnt_call(client,2,xdr_request,&example,            /* this is */
             xdr_returntype,reply,total_timeout))!=RPC_SUCCESS)  /* a micro */
        {                                                  /* which calls the */
           printf("The remote server is either");          /* remote procdure */
           printf(" busy or down.  Please try again ");    /* associated with */
           printf("later.\n");                             /* the client handle*/
           exit(0);                                        /* created above.   */
        }
        clnt_destroy(client);       /* deallocate the space assocated */
                                    /* the client handle.       */
        strcpy(command,"rsh bylands kill -19 "); /* restart the child process*/
        itos(proc_str,reply->childproc_id);      /* of the server's "system  */
        strcat(command,proc_str);                /* co-ordinator".           */
        system(command);
        return (reply);
}


/* The main program body for the first experiment mentioned
   in section 7.2.1.3, which is in the read mode, using the
   un-structured approach.                                        */

main()
{request example1;
 student_record in_info,*out_info;   /* in_info is needed for the write mode */
 struct result *reply;              /* while out_info is required for the   */
 id_info process_ID;                /* read mode.                           */
 char command[cmdlength];
 char proc_str[maxdigits];
 char *row_of_bytes;
   strcpy(example1.application,"\"students.data.bank\"");
   strcpy(example1.command,"\"search\"");
   strcpy(example1.dbname,"\"utility\"");
   strcpy(example1.entry,"\"application2\"");
   strcpy(example1.student,"\"Paul\"");
   strcpy(process_ID.hostname,server_machine);
   process_ID.id=getpid();
   if (callrpc_queue_manager(process_ID)==TRUE)
   {
     strcpy(command,"kill -17 ");   /* signal a sleep signal to itself*/
     itos(proc_str,process_ID.id);  /* and then waiting for service.  */
     strcat(command,proc_str);
     system(command);
     reply=(struct result *)malloc(sizeof(struct result));  /* this is the   */
     reply=callrpc_access_manager(example1);                /* re-start point*/
     if (reply->status==TRUE)
     {
```

```
      out_info=(user_structure *)malloc(sizeof(user_structure));
      out_info=(user_structure *)reply->uval.value;      /* this converts */
      printf("name: %s\n", out_info->name);              /* the returned   */
      printf("age: %d\n",out_info->age);                 /* row of bytes  */
      printf("sex: %c\n",out_info->sex);                 /* back to the   */
      printf("house number: %s\n", out_info->house_nb); /* user's defined */
      printf("street : %s\n", out_info->street);         /* structure.     */
      printf("town : %s\n", out_info->town);
      printf("department : %s\n", out_info->department);
      printf("year : %s\n", out_info->year);
    }
    else
    {
      printf("%s\n",reply->uval.errormsg);
    }
  }
}
```

Appendix E

1) Results obtained from the stack example using the structured approach subject to the experiments mentioned in section 7.2.1.3 of the main thesis are :

experiment(a) - attempts to top an item off the stack when it is empty.

reply: the messsage "stack is empty" is received.

experiment(b) - push the string "hello" into the stack twice.

reply: no message is received.

experiment(c) - repeat experiment(b) above with the same process invoked from two different local machines "ws_lodge1" and "ws_lodge2".

reply: no messsage received.

experiment(d) - retrieve all the inserted strings from the stack.

reply: hello
hello
hello
hello

2) Results obtained from the stack example using the un-structured approach, subject to the experiments mentioned in section 7.2.1.3 of the thesis are identical to those for (1) above.

3) Results obtained from the students data bank example when the structured remote data access methods is applied, subject to the experiments mentioned in section 7.2.2.3. (see also the PS-algol program "utility.class.lib":

experiment(a) - attempts to retrieve a student reocrd from the data bank even when it is empty.

reply: the message "No record yet" is received.

experiment(b) - Using three separate client processes and each of them is used to insert a student record into the data bank. The contents of these three records are:

(i) name: Alan Tam
age: 35
sex: M
house.nb: 20
street: Steavenson
town: Bowburn
department: Physics
year: 3

(ii) name: Kenny Luk
age: 29
sex: M
house.nb: 29
street: Silver

```
                              town: Durham
                              department: Economic
                              year: 2

                        (iii) name: Sammy Mui
                              age: 22
                              sex: F
                              house.nb: 19
                              street: Steavenson
                              town: Bowburn
                              department: Business School
                              year: 3

  reply: no error message is received from any of the three processes.

experiment(c) - search the three records inserted by experiment(b) above

  reply1: Alan Tam
          35
          M
          20
          Steavenson
          Bowburn
          Physics
          3

  reply2: Kenny Luk
          29
          M
          29
          Silver
          Durham
          Economics
          2

  reply3: Sammy Mui
          22
          F
          19
          Steavenson
          Bowburn
          Business School
          3

experiment(d) - update the contents of record(i) and record(ii)
                with the age field changed from 35 and 29 to 36 and 30
                respectively.

  reply: no error message is received.

experiment(e) - retrieve all the information about Alan Tam one by one.

  reply1: Alan Tam

  reply2: 36
```

reply3: M

reply4: Steavenson

reply5: Bowburn

reply6: Physics

reply7: 3

experiment(f) - modify the record of Kenny Luk by replacing the content of the year field from 2 to 3.

reply: no error message

experiment(g) - Choose the record about Sammy Mui as the target, a process is trying to read the content of the age field while another process tries to increase that value by 1 at the same time. This experiment is repeated 6 times.

reply received by the first process:

try1: 22

try2: 24

try3: 25

try4: 25

try5: 26

try6: 28

The results of this particular type of experiment depends heavily on the arrival time of the two processes which will, in turn, determine their position inside DCSUNIX's FIFO queue.

Experiment(h) - retrieve all records form the data bank, followed by another search operation.

reply1: Alan Tam
36
M
20
Steavenson
Bowburn
Physics
3

reply2: Kenny Luk
30
M

```
                29
                Silver
                Durham
                Economics
                3

        reply3: Sammy Mui
                28
                F
                19
                Steavenson
                Bowburn
                Business School
                3

        reply4: An error message, "No record yet", is received.
```

(4) Results obtained from the student data bank example using the un-structured approach have the same results as those in (3) presented previously, except that experiments (e), (f), (g)described in (3) are impossible to be carried out by this approach.

Appendix F

Total number of data transmitted (in bytes)	Average CPU time elapsed (in 1/60 seconds)
80	6.2
1024	39.5
2048	69.2
3072	103.1
4096	150.8
5120	178.9
6144	220.1
7168	255.2
8192	300.6
9216	420.0
10240	460.8

Table 1 Results obtained from the stack example using the un-structured access approach

Total number of data transmitted (in bytes)	Average CPU time elapsed (in 1/60 seconds)
80	12.3
1024	79.8
2048	138.3
3072	206.8
4096	300.0
5120	413.2
6144	460.9
7168	530.2
8192	600.1
9216	840.7
10240	930.1

Table 2 Results obtained from the stack example using the structured access approach

Total number of data transmitted (in bytes)	Average CPU time elapsed (in 1/60 seconds)
80	7.1
1024	40.3
2048	65.1
3072	120.0
4096	140.6
5120	200.9
6144	230.8
7168	260.2
8192	300.1
9216	400.8
10240	410.8

Table 3 Results obtained from the data bank example using the un-structured access approach

Total number of data transmitted (in bytes)	Average CPU time elapsed (in 1/60 seconds)
80	13.0
1024	72.5
2048	117.2
3072	220.6
4096	280.1
5120	361.6
6144	415.4
7168	470.2
8192	540.2
9216	721.3
10240	820.3

Table 4 Results obtained from the data bank example using the structured access approach

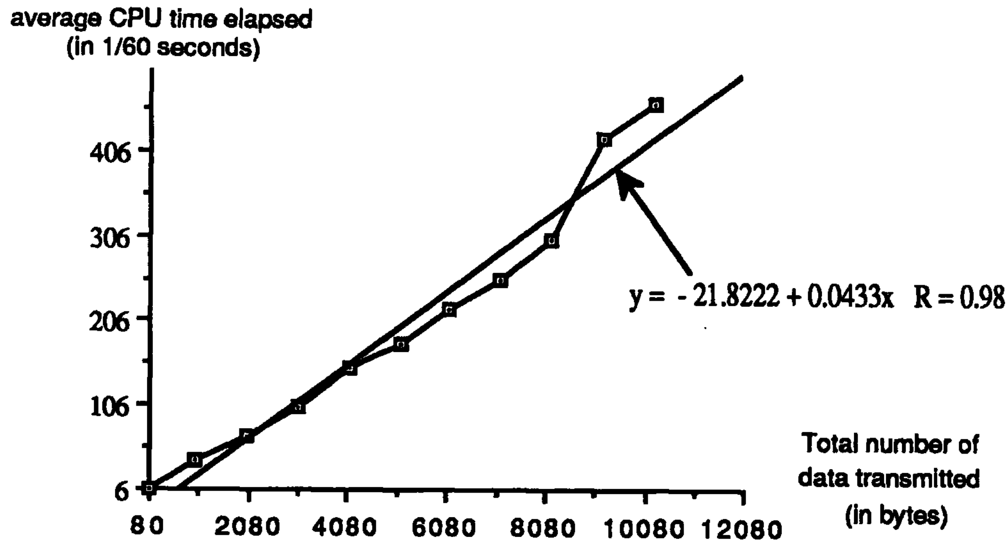


Figure 1 Performance graph of the stack example using the un-structured approach

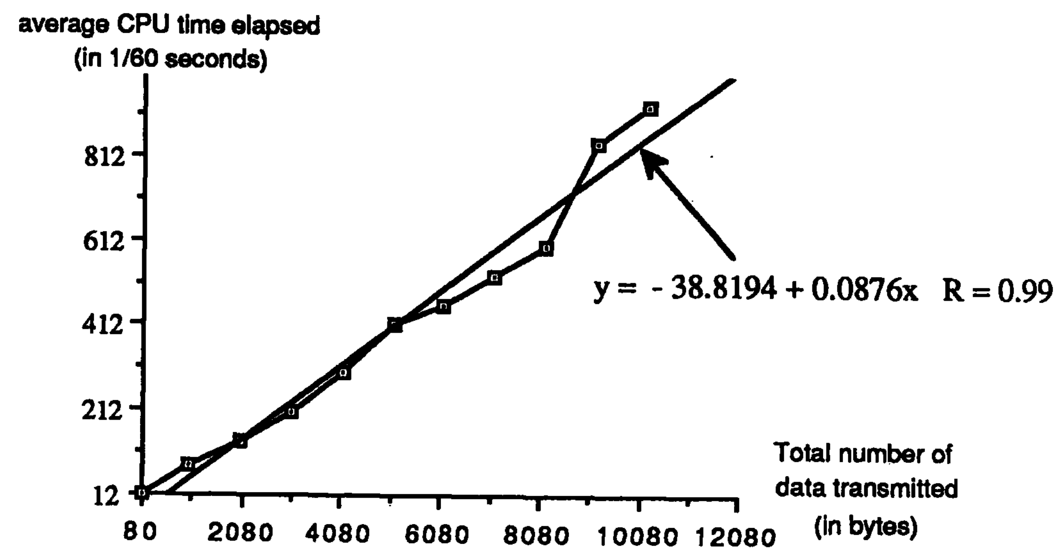


Figure 2 Performance graph of the stack example using the structured approach

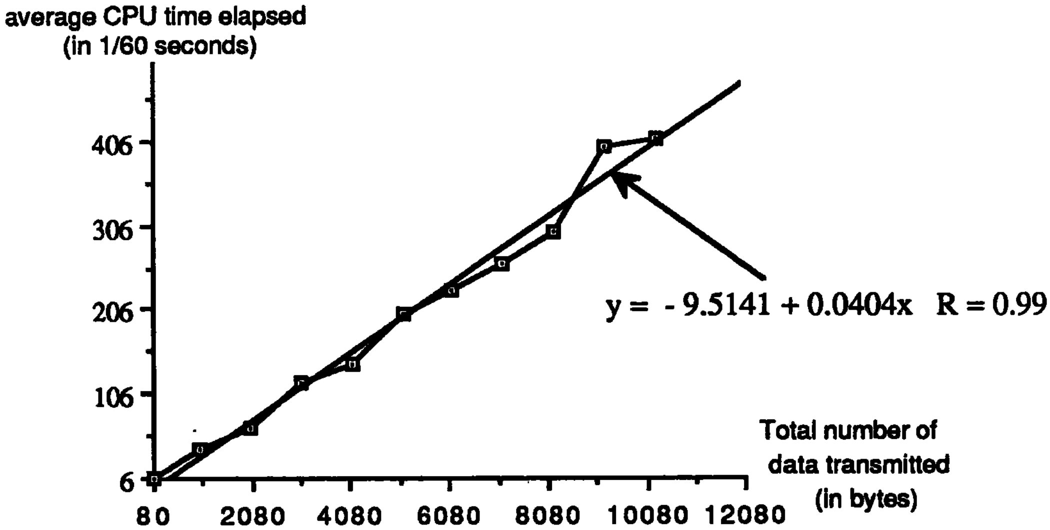


Figure 3 Performance graph of the data bank example using the un-structured approach

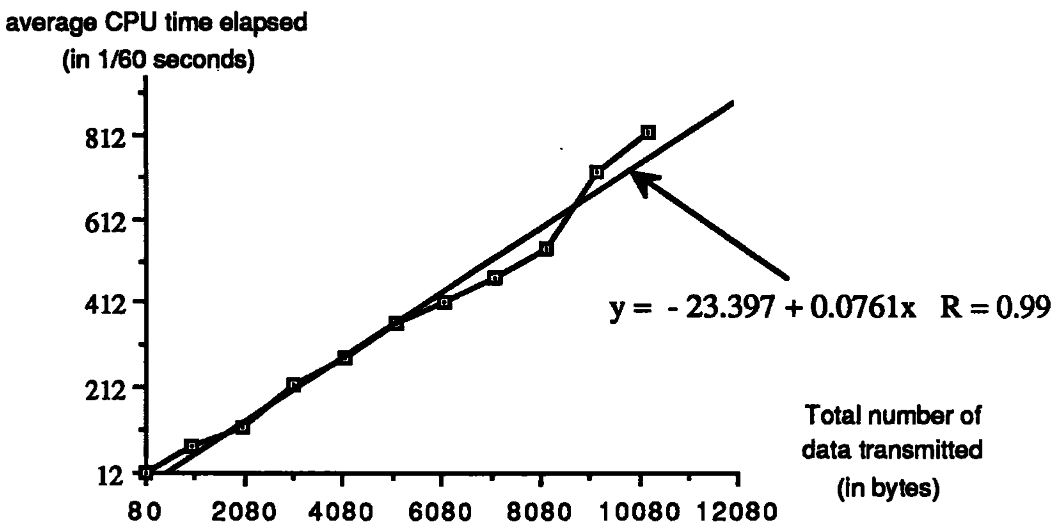


Figure 2 Performance graph of the data bank example using the structured approach

References

[1] NELSON, B. 1981. "Remote Procedure Call". Ph.D thesis, Tech. Report. CSL-79-3, Xerox Palo Alto Research Centre.

[2] JONES, A. and GEHRINGER, E. 1980(July). "The CM* Multiprocessor: A Research Review". CS-80-131, Department of Computer Science, Carnegie-Mellon University.

[3] DAGLESS, E. 1977. "A Multimicroprocessor- CYBA-M". IFIP North Holland.

[4] DALAL, Y.K. 1977. "Broadcast protocols in packet switched computer networks". Ph.D thesis, Computer Science Dept, Standford University, Standford, California.

[5] NEEDHAM , R.M., and HERBERT, A.J. 1982. "The Cambridge Distribued System". Addison- Wesley, Reading Mass.

[6] POPEK, G., WALKER, B., CHOW,J., EDWARDS, D., KLINE, C., RUDISION, G., and THIEL, G. 1981. "Locus: A network transparent high reliability distributed system". In Proceedings of the 8th Symposium on Operating Systems Principles (Pacific Grove, Calif., Dec 14-16) ACM. New York, pp. 160-168.

[7] CHERITION, D.R. 1984(Apr). "The V Kernel: A software base for distributed systems". IEEE Softw. 1, pp. 19-42.

[8] ALMES, G.T., BLACK, A.P., LAZOWSKA, E.D., and NOE, J.D. 1985(Jan). "The Eden System: A technical review". IEEE Trans.Softw. Eng. SE-11, pp. 43-59.

[9] MILLSTEIN, R.E. 1977. "The national software works". In Proceedings of the ACM Annual Conference. ACM, New York, pp. 44-52.

[1 0] TANENBAUM, A.S. and RENESSE, R.V. 1985(Dec). "Distributed operating systems". ACM Computing Survey, vol 17, No 4, pp. 419-471.

[11] LISKOV, B. 1982(May). "On Linguistic support for distributed programs". IEEE Trans.Soft.Eng. SE-8, pp. 203-210.

[12] DENNIS, J. B., and VAN HORN, E. C. 1966(March). "Programming semantics for multi-programmed computations". Comm. ACM, 9, 3, pp.143-155.

[13] ZIMMERMANN, H. 1980(Apr). "OSI reference model- The ISO model of architecture for open systems interconnection". IEEE Trans. Comm. COM-28, pp. 425-432.

[14] SALTZER, J.H., REED, D.P., and CLARK, D.D. 1984(Nov). "End-to-end arguments in system design". ACM Trans.Compt.Syst. 2, 4, pp. 277-278.

[15] SVENTEK, J., GREIMAN, W., O'DELL, M. and JANSEN, A. 1983. "Token ring local networks- A comparison of experimental and theoretical performance". Lawrence Berkeley Lab. Rep. 16254.

[16] MANCINI, L. V., and SHRIVASTAVA, S. K. 1988(June). "Fault-tolerance reference-counting for garbage collections in distributed systems". Technical Report Series, No. 259, Computing Laboratory, University of Newcastle Upon Tyne.

[17] SHOCH, J. F. 1978. "Inter-network Naming, Addressing and Routing". Compcon, IEEE.

[18] GOLBERG, A., ROBSON, D. 1983. "Smalltalk-80: the language and its implementation". Addison-Wesley, Reading, MA.

[19] WITTIE, L.D., and VAN TILBORG, A.M. 1980(Dec). "MICROS: A distributed operating system for MICRONET, a reconfigurable network computer". IEEE Trans.Comp. C-29, pp. 1133-1144.

[20] STONE, H. S. 1977. "Multiprocessor scheduling with the aid of network flow algorithms". IEEE Trans.Soft.Eng. SE-3, pp. 88-93.

[21] OUSTERHOUT, J.K. 1982. "Scheduling techniques for concurrent systems". In Proceedings of the 3rd International Conference on Distributed Computing Systems. IEEE, New York, pp. 22-30.

[22] BARAK, A., and SHILOH, A. 1985(Sept). "A distributed load-balancing policy for a multicomputer". Soft.Pract.Exper. 15, pp. 901-913.

[23] STANKOVIC, J. A., and SIDHU, I. S. 1984. "An adaptive bidding algorithm for processes, clusters and distributed ups". In Proceedings of the 4th International Conference on Distributed Computing Systems. IEEE, New York, pp. 49-59.

[24] CHANDY, K.M., MISRA, J., and HAAS, L.M. 1983(May). "Distributed deadlock detection". ACM Trans.Comp.Syst. 1, 2, pp. 145-156.

[25] POWELL, M.L., and PRESOTTO,D.L. 1983. "A reliable broadcast communication mechanism". Oper.Syst.Rev (ACM) 17, 5, pp. 100-109.

[26] AVIZENINS, A., and CHEN, L. 1977. "On the implementation of N-version programming for software fault-tolerance during execution". In Proceedings of the International Computer Software and Applications Conference. IEEE, New York, pp. 149-155.

[27] GRAY, J. N. 1979. "Operating system: An Advanced course". Springer-Valag, New York, pp. 393-481.

[28] LAMPSON, B.W. 1981. "Atomic transactions. In Distributed Systems-Architecture and Implementation". Springer-Verlag, Berlin and New York, pp. 246-265.

[29] ESWARAN, K. P., GRAY, J.N., LORIE, J.N., and TRAIGER, I. L. 1976(Nov). "The notions of consistency and predicate locks in a database system". Comm.ACM, 19, 11, pp. 624-633.

[30] BIRRELL, A.D. LEVEN, R., NEEDHAM, R.M. and SCHROEDER, M. 1982(Apr). "Grapevine: An exercise in distributed computing". Comm. ACM, 25, 4, pp. 260-274.

[31] TANENBAUM, A.S., MULLENDER, S.J., and VAN RENESSE, R. 1986. "Using sparse capabilities in a distributed operating system". In Proceedings of the 6th International Conference on Distributed Computer Systems. IEEE, New York, 1986. pp. 558-563.

[32] CHERITON, D.R. and ZWAENEPOEL, W. 1983. "The distributed V kernel and its performance for diskless workstations". In the Proceedings of the 9th Symposium on Operating System Principles. ACM, New York, pp. 128-140.

[33] WILKES ,M. V. and WHEELER ,D. J. 1979. "The Cambridge digital communication ring". Proc. Local Area Communications Network Symp, Boston, Nat.Bur.Standards special publications.

[34] WILKES ,M. V. and NEEDHAM , R. M. 1979. "The Cambridge CAP computer and its operating system". Operating and programming System Series. Elsevier, North Holland.

[35] DIXON, J. 1980 (Oct). "The Cambridge File Server". ACM Operating System Review 14(4), pp. 26-35.

[36] TANENBAUM, A.S. 1981. "Computer Networks". Prentice/Hall International Editions.

[37] GARNETT, N.H. and NEEDHAM, R.M. 1980(Oct). "An Asychronous Garbage Collector for the Cambridge File Server". ACM Operating System Review 14(4), pp. 36-40.

[38] STURGIS, H., MITCHELL, J. and ISRAEL, J. 1980 (July). "Issues in the Design and Use of a Distributed File System". ACM Operating System Review 14(3), pp. 55-69.

[39] RANDELL, B. 1983 (May). "Recursively Structured Distributed Computing Systems". Report SRM/346, Computing Laboratory, University of Newcastle Upon Tyne.

[40] DEEN, S.M. 1986. "Fundamental of Data Base Systems". Macmillan Press.

[41] MCFADDEN, F.R. and HOFFER, J.A. 1985. "Data Base Management". The Benjamin/Cummings Publishing Company.

[42] CLARK, J.D. 1980. "Database Selection, Design and Administration". New York, Praeger Publishers.

[43] BRADLEY, J. 1982. "File and Database Techniques". HRW, University of Calgary.

[44] CULLINANE 1975. "Integrated Database Management System (IDMS) Brochure". CULLINANE Corporation.

[45] TAYLOR, R.W. and FRANK, R.L. 1976(March) "CODASYL Database Management Systems". ACM Computing Surveys, Vol 8, No.1, pp. 67-103.

[46] CODD, E.F. 1970(June). "A Relational Model of Data for Large Shared Data Banks". Comm.of ACM, vol 13, No.6, pp. 377-387.

[47] Relational Technology Inc. 1984. "Introduction to INGRES". Relational Technology Inc., California.

[48] GROSS, J.M., JACKSON, P.E., JOYCE, J., and MCGUIRE, F. A. 1980. "Distributed database design and administration". Cambridge University Press, pp. 285-296.

[49] HOROWITZ, E. 1985. "Programming Languages: A Grand Tutor; a collection of papers edited by E. Horowitz". 2nd Edition, Computer Software Engineering Series, Rochville, Computer Science Press.

[50] MARTIN, J.J. 1986. "Data Types and Data Structures". Prentice/Hall International.

[51] THOMAS, P.G., ROBINSON, H.M. and EMMS, J.M. 1988. "Abstract Data Types : their Specification, Representation and Use". Oxford Applied Mathematics and Computing Science Series.

[52] TEXEL, P., P. 1986. "Introductory Ada". Wadsworth, Belmont, California.

[53] WIRTH ,N. 1985. "Programming in Modula-2". Springer-Verlag, Berlin.

[54] LISKOV ,B., ATKINSON ,R. 1981. "Lecture Notes in Computer Science: CLU Reference Manual". Springer-Verlag, Berlin, Heidelberg, New York.

[55] BIRTWISTLE, D. MYHRHANG and NYGAARD. 1979. "Simula-Begin" (2nd Edition).

[56] STUART, F. 1970. "Fortran Programming". John Wiley and Sons Inc.

[57] RANDELL, B. and RUSSELL, L.J. 1964. "Algol-60 Implementation- the translation and use of Algol-60 programs on a computer". Academic Press.

[58] RUSTON, H. 1978. "Programming with PL/I". McGraw-Hill Book Company.

[59] ATKINSON, L.V. 1980. "Pascal Programming". John Wiley and Sons Inc.

[60] WIJNGAARDEN , V. 1975. "Revised report on the Algorithmic language Algol-68". Acta. Information.

[61] AMERICAN NATIONAL STANDARD INSTITUTE. 1985. "American National Standard Programming Language COBOL". ANSI.X3-1985.

[62] WEINREB, D., and MOON, D. 1981. "LISP Machine Manual". Symbolic Inc., Cambridge, MASS.

[63] MILNER, R. 1984. "A proposal for standard ML". In Proceedings of the Symposium on LISP and Functional Programming (Austin, Texax. Aug 6-8). ACM, New York, pp. 184-197.

[64] CARDELLI, L. and WEGNER, P. 1985(Dec). "On understanding Types, Data Abstraction and Polymorphism". ACM Computing Survey, Vol 17, Number 4, pp. 471-523.

[65] ROBINSON, J. A. 1965(Jan). "A machine-oriented logic based on the resolution principle". ACM 12, pp. 23-49.

[66] ATKINSON, M.P. and BUNEMAN, O.P. 1987(June). "Types and persistence in database programming languages". ACM Computing Survey, Vol 19, Number 2.

[67] ATKINSON, M. P., and MORRISION, R. 1985. "Types, binding and parameters in a persistent environment". In Proceedings of the Appin Workshop on Data Types and Persistence Research Report 16. Persistent Programming Group, Dept of Computing Science, University of Glasgow, Glasgow, Scotland.

[68] MORRISION, R. 1979. "S-algol Language Reference Manuals". Technical Report Rep. CS/79/1, Dept of Computer Science, University of St.Andrew, St.Andrew, Scotland.

[69] ATKINSON, M.P. and MORRISION, R. 1985. "Procedure as persistent data objects". ACM Trans.Prog.Lang.Syst. 7, 4, pp. 539-559.

[70] MORRIS, J.H. 1973. "Protection in programming languages". Comm.ACM. 16, 1, pp. 15-21.

[71] ZILLES, S.N. 1973. "Procedural Encapsulation. A linguistic protection technique". In ACM SIGPLAN NOTE 8, 9.

[72] STRACHEY, C. 1967. "Fundamental Concepts in Programming Languages". Oxford University Press, New York.

[73] RELATIONAL TECHNOLOGY INC. 1987. "INGRES for IBM-PCs and compatibles - Reference Guide". Relational Technology Inc.

[74] ATKINSON, M.P, BAILEY, P., COCKSHOTT, W.P., CHISHOLM, K.J. and MORRISON, R. 1986. "PS-algol Reference Manual". Fourth Edition, Persistent Programming Research Report 12, University of St. Adrews, Scotland.

[75] MORRISON, R., CARRICK, R., and COLE, A.J. 1986. "An Introduction to PS-algol Programming". Second Edition, Persistent Programming Research Report 31, University of St. Adrews, Scotland.

[76] METCALF, R. and BOGGS, D.R. 1976(Jun). "Ethernet : Distributed Packet Switching for Local Computer Networks". Comm.ACM, Vol 19, No. 19, pp. 395-404.

[77] SUN MICROSYSTEMS INC. 1986(Feb). "Unix Interface Reference Manual". SUN Microsystems Inc.

[78] SWINEHART, D., MCDANIEL, G. and BOGGS, D. 1979 (Dec). "WFS: A simple shared file system for a distributed environment". In Proceedings of the 7th ACM Symposium on Operating System Principles. Oper.Syst.Rev. Vol 13, No.4, pp. 9-17.

[79] DELLAR, C. 1982. "A file server for a network of low-cost personal microcomputers". Soft.Prat.Exper. 12, pp. 1051-1068.

[80] BROWN, M.R., KOLLING, K. and TAFT, E.A. 1984(Oct). "The Alpine file system". Technical Report CSL-84-4, Xerox Palo Alto Research Centre, Palo Alto, California.

[81] SATYANARAYANAN, M. 1984. "The ITC project: An experiment in large-scale distributed personal computing". In Proceedings of the Networks 84 Conference.

[82] MCKENDRY, M.S. AND HERLIHY, M. 1986 (Jun). "Time-driven orphan elimination". In the 5th Symposium on Reliability in Distributed Software and Database Systems, Los Angeles, California, pp.42-48.

[83] HERMAN, G. 1986(Oct). "CD-ROMS: the future of mass storage?". Electronics Today, pp.22-25.

Glossary

Abstract data type : It is a collection of data together with the operations that can be carried out on that data.

Abstraction : A way to detach level of concerns from programmers.

Access time : The time the computer takes to obtain a word from a storage device.

Activity distribution curve : A graph which shows the number of a specific typed records involved at a particular time.

Association : The relationship between two entities.

Attribute : An attribute (in database terminology) is a property of an entity that is chosen to record.

Axioms : Generally accepted truth or principles.

Bandwidth : It is the frequency range in a system that are available for transmission.

Baud : The number of signal changes per second.

Bottom-up : Constrast to the method of top-down in which all the specific details are planned first, eg. the input, output and data structures of a program, before the overall assembly of the program is started.

Capability : A unique name used to control access to an object (file, procedure etc).

Centralised network : Network with a topology such that all nodes are connected to a single node.

Client : Process which accesses a resource or an object in a server. The server and client will be assumed in different nodes.

CLU Iterator : A unit to achieve control abstraction in CLU.

CLU procedure : A unit to achieve procedural abstraction in CLU.

Codasyl : Short for the Conference of Data System Languages which is a voluntary group of individuals who represent hardware and software vendors, universities and major developers and users of data processing systems.

Cohesion : A property of an individual module. A module is said to be strongly cohesive if it is responsible for a single function; however, if a module that is responsible for two or more distinct functions, it is said to be weakly cohesive.

Conceptual database model : A data model which concerns only the overall design architecture of a database.

Context : A list of bindings where a binding is defined as an association between a name and its identity.

Context swapping : The focus of control for sharing a processor.

Coupling : It is an issue concerned with inter-connections between modules. A module X is said to be tightly coupled to another module Y if it has many dependencies on Y, or if any of those dependencies are complex. On the other hand, X is loosely coupled to Y if it has only a few, simple dependencies on Y.

Data : Raw materials eg. numbers, characters.

Database : A collection of stored operational data used by the application systems of some particular enterprise.

Datagram : Similar to virtual circuit but in this case the network layer accepts messages from the transport layer and attempts to deliver each one as an isolated unit. Messages may arrive out of order or not at all.

Data item : It is the smallest unit of data that has meaning to a user.

Data processing : The process of collecting all items of source data together and converting them into information.

Decentralised network : A distributed network of centralised sub-networks.

Desktop : A user interface based on the idea of several documents lying on a desk, the documents are represented by a series of windows drawn on the screen of a personal computer.

Distributed computing systems : Systems that look to their users as an ordinary centralised system but run on multiple, independent central processing units.

Distributed network : Network in which all the nodes have multiple connections to other nodes.

Entity : An entity is a person, a place, an object, an event or a concept about which an organization wishes to record.

Exceptions : Variables which can be set to indicate the occurrence of unusual events.

Expressive power: The ability to perform arbitrary computations in programming languages.

File server : A repository where files can be stored and which provides an index address for files contained in it.

Filestore : A repository for data, providing a mnemonic (user-arbitrary) naming scheme for files.

Firmware : In the context of microprogramming, a term applies to any resident programs in a ROM , i.e. it cannot be altered.

First class values : They are legal values that can be passed and returned from functions and stored in data structures.

Formal language : A language with a precisely agreed set of rules governing its use such that a statement in the language has exactly one meaning.

Half-duplex : Data which can travel in both directions, but not simultaneously.

Host : This is usually taken as a user computer connected to a network which either provides services for users or is concerned with activities other than those purely networking ones.

Full-duplex : Data which can travel in both directions at once.

Icon : A graphical representation of an object, a concept or a message.

Implementation : A program code which carries out the operations of an A.D.T.

Implementation error : Software failed to meet the formal specification.

Information : Data which has meaning eg. name, address, age etc.

Information hiding : A mechanism to separate specification of a subprogram (i.e. the parameters of the subprogram) from the body of the subprogam.

Inheritance : It means using already existed software.

Interface : Conventions for communication across adjacent layers.

Kernel : For the purpose of htis thesis, the term kernel is taken to mean the kernel of an operating system. A kernel consists of the body of code that is intensively and commonly used by all programs at higher levels as if it were an extension of a machine. Functions normally found in the kernel are: interrupt handling, I/O support, dispatching (the role of deciding which process to run next), memory management and possibly file management.

Locking : A mechanism which provides the exclusive access rights of data object(s).

Message : The smallest unit of data that must be sent and received between a pair of correspondents for a meaningful action to take place.

Module : A program construct which has a name and a well-defined boundary.

Multiprogramming : A technique whereby several programs are placed in main memory at the same time, giving the illusion that they are being executed simultaneously but in fact, they areb being executed consecutively.

Namespace : A collection of contexts which are mutually accessible.

Network distributed system : System which consisted of a collection of computers, each one has different responsibility and all tied together by a local network such as Ethernet.

Node : A point of convergence of communication paths in a network.

Object : From the object-oriented programming point of view, it is treated as an abstract data type.

Operating system : A program that controls the resources of a computer and provides its users with an interface or virtual machine that is more convenient to use than the bare machines. This term has been often used to refer to all manufacturer-supplied software such as I/O programs and compilers.

Orthogonality : The facility possessed by a programming language to apply its constructs either indivdually or in any combination without imposing restriction.

Packet : The smallest unit of message involved during a communication.

Packet switching : A method of data transmission where information is sent in a packet that includes an address at the front; this address is used to route the packet to its destination via a transmission protocol such as X25.

Page : A fixed-length storage unit.

Page swapping : The act of exchanging a page in main memory with a page in the secondary store such as a disc.

Paging : The process of transmitting required pages from the secondary store into the main memory for execution.

Procedural decomposition : A technique to sub-dividing programs into procedures and functions.

Protocol : An interface between two distributed (remote to each other) co-operating modules of the same level.

Representation : It describes the data structures which will be used in subsequent implementation of an A.D.T.

Response time : The time taken for a system to react to a user input.

Robustness : The ability of software systems to function even in abnormal conditions.

Routing : Algorithm to send message to the destination host across the network.

Semaphores : Low level synchronization primitives used by concurrent processes to send signals to each other during a communication session.

Server : A node where the representation for a given object type and the operations on this representation are implemented.

Service : An abstract specification of generic primitives, their results and sequencing.

Simplex : Data which can only travel in one direction.

Software engineering : The subject of finding ways to build quality software.

Software life cycle : A term used to describe the stages involved during the development and usage of a large software system.

Software maintenance : The process of modifying a program after it has been delivered and is in use.

Specification : A definition which is independent of representation or implementation details. It describes the operations of an A.D.T. independently without knowing how those operations are carried out.

Specification error : Specification deos not correctly model what the customer wanted.

Sub-network : A network which is itself a component of a network.

Throughput : The number of useful data bits per second that reach the receiver in some time interval.

Transfer rate : The number of bytes transmitted per second.

Top-down : A methodology to tackle a problem by breaking it down into sub-problems and these sub-problems will be broken further into smaller sub-problems until the entire problem has been reduced to a collection of easily solved sub-problems.

Type : Generally, it refers to the internal representation of an object.

Type consistent : The type of an expression remains un-changed during the whole execution of a program.

Virtual circuit : During the transmission of a message in the ISO OSI reference model, the network layer provides the transport layer with a perfect channel (no errors), therefore all packets delivered in order.

Workstation : A machine which is usually connected to one or more mainframes but with its own processor, memory, a bit-mapped display and sometimes a disc. Therefore, a workstation provides a place for users to work.