

BROWN UNIVERSITY
Department of Computer Science
Master's Project
CS-94-M14

“NIS+NSI
A Name Service Adjunct to SUN's NIS+”
by
Shamsi Moussavi-Aghdam

NIS+NSI

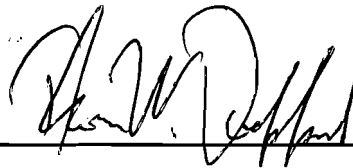
A Name Service Adjunct to SUN's NIS+

Shamsi Moussavi-Aghdam

Department of Computer Science
Brown University
Providence RI

Submitted in partial fulfillment of the requirements for the
Degree of Master of Science in the Department of
Computer Science at Brown University

May 1994

A handwritten signature in black ink, appearing to read 'Thomas W. Doepner', is written over a solid horizontal line.

Professor Thomas W. Doepner
Advisor

NIS+NSI

A Name Service Adjunct to SUN's NIS+

Shamsi Moussavi-Aghdam
Department of Computer Science
Brown University
May 1994

Abstract

NIS+NSI is designed to complement SUN's NIS+ with useful features from OSF DCE's NSI such as search to find an array of host and search for one host at random. In addition to these features, NIS+NSI provides a search interface to find the most idle workstations on the network. It offers clients an easy way of finding hosts, especially for parallel application programmers. Clients need only supply a name for the starting point for a search. The starting point can even be a default name. NIS+NSI takes advantage of NFS for its underlying database and uses Quahog, a distributed system developed at Brown, to get the list of idle workstations. This paper begins with a general overview of some existing name services and their cons and pros. It then continues with a description of NIS+NSI and ends with a parallel application example which utilizes NIS+NSI.

INTRODUCTION:

In the current world of distributed computing, name services play an important role. Since name services are an important part of the distributed computing model, they have a large effect on application performance. Name services would be even more effective if they could provide additional services. The purpose of this project is to add new and useful features to SUN's RPC Network Information Service (NIS+).

The design of the NIS+NSI makes use of knowledge of both SUN's NIS+ and OSF DCE's NSI. The availability of NIS+ and effectiveness of NSI were two important components in the creation of NIS+NSI. This project has added NSI's efficient search interfaces in finding hosts to SUN's widely used RPC. Consequently, NIS+NSI is literally a combination of NIS+ and NSI. In the NIS+NSI environment, servers export their services and clients import them. NIS+NSI returns to the client a host or an array of hosts, depending on the type of request. As opposed to NIS+, in the NIS+NSI environment, clients are not required to explicitly give a host name for a RPC call. NIS+NSI also gives clients the opportunity to make smart choices for applications with heavy computation by searching for the most idle workstation. A more detailed description of NIS+NSI will be given in Section 4.

In order to gain further understanding of the above concept, a discussion on name services is provided in Sections 1, 2, and 3 to answer the following questions: What are name services?

Why are name services important?

Why should one bother to implement additional Name services features for a product which has already a name service?

Section 4 describes NIS+NSI and Section 5 contains an example of using NIS+NSI in the context of a parallel application using SUN RPC.

1. NAME SERVICE'S FUNCTIONS

In general, name services' main responsibility is to provide information for users, applications, and workstations which need to communicate on the network. Name services take the burden of storing all network information on each machine.

Thus it saves a substantial amount of space. Without name services, each machine has to keep a list of all other machines' addresses and the related information that is useful for a distributed computation. Name services are the primary source of information about different machines on the network. Certainly, addresses are not the only information name services store. Machines' names are another part of information that must be kept by all name services. There are many advantages for having names instead of addresses in communication protocols such as: the binding of name to address is delayed to the time of service, same service name can actually have multiple providers of the service and the network administrators are able to transparently add or remove a provider. A closer look at a few different existing name services shows the domains of information that are kept and serviced by different name services:

1.1 DNS

Domain Name Service (DNS) is an application layer protocol that is part of the standard TCP/IP protocol suite. DNS was originally developed to provide machines' names instead of Internet addresses across the network. The hierarchy of domains form the DNS's *namespace*, where a domain is organized from a group of machines. Each domain stores the names and addresses of all machines in the group. Educational, commercial, and international organizations are a few examples of domains in DNS's namespace. *Name servers* provide the DNS services. DNS provides naming services, both translating from hostnames to IP addresses and vice versa. DNS obtains and provides information about hosts on a network, and provides naming services between hosts within the local domain and across domain boundaries. Another DNS service is to assist mail agents to deliver mail across the Internet. [1]

1.2 NIS & NIS+

The primary goal of developing SUN's Network Information Service (NIS) was to provide a centralized control over network information and thereby make network administration more manageable. NIS stores a collection of network information. In addition to names and addresses of machines, NIS stores information about network and its services. NIS keeps various information regarding networks such as: names and addresses of networks, list of known RPC program name and number, list of available Internet services, and list of NIS servers in the *namespace*. NIS' *namespace* is referred to this collection of information. The NIS namespace is structured as a flat arrangement of domains. NIS uses a client-server model, with replicated NIS servers providing service to NIS clients.

NIS+, the new version of NIS, was developed in response to the demand for more reliable name services, due to ever growing client-server networks. The centralized, flat structure of NIS didn't match the demands of growing domains; therefore, NIS+ adopted the hierarchical model of DNS. Unlike DNS, however, NIS+ retains NIS's ability to handle a variety of network information. Furthermore, NIS+ provides a security scheme to protect the information in the *namespace* as well as the structure of the *namespace* itself from unauthorized access. [1]

1.3 DCE

NSI (Naming Services Interface), as a part of the RPC package of OSF's (Open Software Foundation) DCE (Distributed Computing Environment), is yet another name service model. DCE RPC is integrated with the DCE Directory Service component, which facilitates the search for finding the server's location. The DCE *namespace* used by DCE Directory Service has a hierarchical structure. DCE Directory Services make use of two global directory services: GDS* and DNS, which are used to name *cells*. A *cell* is a collection of DCE machines that work together and are administered as a unit. It is at the *cell* level of the structure that CDS (Cell Directory Service), yet another DCE Directory Services component, is employed. CDS stores the information about resources within a *cell*.

NSI uses CDS as its database. Each CDS entry has numerous CDS attributes, but NSI uses only four different attributes: binding, object, group, and profile attributes. Therefore, an entry in NSI can be one of the following:

- server entry Contains a binding and an object attribute, making it suitable to hold the necessary binding information for a single server.

- group entry Contains a group attribute.

- profile entry Contains a profile attribute. [2]

Here *binding attribute* contains one or more sets of binding information; *object attribute* contains one or more *object ID* (called UUID); *group attribute* contains references to one or more of other NSI entries; and *profile attribute* also contains one or more references to other NSI entries including another profile entry.

The advantage of NSI over SUN's NIS or NIS+ is the range of services it provides. Servers advertise themselves to clients by exporting their services to NSI,

.....
* GDS, Global Directory Service, a DCE Directory Service component.

and clients query NSI to find and bind to servers. NSI, as part of the DCE RPC API, gives clients the choice of different approaches to finding a compatible server. Among the library routines provided by NSI are the following:

rpc_ns_binding_import_next()

Searches the NSI database and returns one binding handle for a compatible server.

rpc_ns_binding_lookup_next()

Searches the NSI database and returns a list of binding handle of all the compatible servers up to a maximum number.

rpc_ns_binding_select()

Selects one binding handle at random from the list. [2]

2. THE IMPORTANCE OF NAME SERVICES

Name services perform a unique service for the network. It would be rather cumbersome to take advantage of network facilities without having name services at the network level. The two important features of name services are:

1- Time factor:

Name services improve application performance by providing the information they need fast.

2- Space factor:

Name services prevent waste of space on machines, because only a few specific machines store the name services information; thus, all other machines' disk space remains free for other purposes.

3. IMPROVEMENT OF NAME SERVICES

It is the nature of human beings to constantly strive for improvement. More efficient Name Services are needed with the passage of time. In order to answer the demand of ever-evolving networks, improved and advanced features must be added to Name Services. Users are expecting more from name services than before. Name services can no longer be a database of addresses and names, with support for a few queries. The growth of networks requires constant improvement of name services. NIS+ was developed to combine the best aspects of two already existing name services and to add more advanced features, such as security.

This project is an attempt to validate the necessity of improved name services.

4. NIS+NSI

NIS+NSI is a name service adjunct to NIS, called NIS+NSI because it complements the SUN's NIS+ name services with features found in DCE's NSI. It is a name service adjunct, rather than a complete name service, because it lacks a database to keep all sorts of information and various interfaces for obtaining information. But it is a necessary part of a name service, because it provides significant services that NIS lacks. The goal of NIS+NSI is to provide an interface for a number of search facilities similar to DCE NSI's operations. SUN's RPC requires the clients to provide the remote host name. NIS+NSI relieves this constraint from SUN RPC's clients by providing the search interface. As is the case in DCE's NSI, servers must export RPC interfaces to NIS+NSI. However, there is no notion of *namespace* in NIS+NSI. No underlying database for storing the information exists in NIS+NSI. Instead, NFS (SUN's Network File System) handles this responsibility. A file is created for each interface exported with the information about the RPC interface and server addressing information. The search operations eventually open the file and get the information. Various number of binding information can be returned, depending on the client's type of request from NIS+NSI.

4.1 NIS+NSI ENTRIES

In addition to host entry files, there are profile and group entry files. These profile and group files are created externally and form a hierarchical structure: the host files represent the leaves of the tree and the root can be either a group file or profile file. The hierarchical file structure makes it easier to search for any host which provides the particular interface ID and version, without having to specifically know the host's name. However, the client must supply a file name as the starting point for the search. But the file name can be the name of a profile file, group file, host file, or a default name. The default name is a NIS+NSI environment variable, which is the name of a profile file containing all of the different interface IDs and versions. Furthermore, the client can also start a search at a level closer to the host. The client has the option of giving the group or the profile file name, if she/he knows that would lead to the desired host or hosts.

Profile files hold the information about other profile files, group files and host files which have common characteristics, e.g. they all are located in one building. A profile file contains a series of entries, where each entry consists of the interface ID,

interface version number, and the file name. The file name is the name of the file, which contains further search direction if the corresponding interface id and version number match the requested interface id and version. The file name is one of the following: another profile file, a group file or a host file. The interface ID and version number of the elements in this file are examined to find the compatible entry, from which the search is continued.

Group files hold the information about hosts or groups which have common characteristics, e.g. the same RPC interface ID. The format of a group file is an entry number and again a file name. The file name plays the same role as the file name in profile file, but can be the name of either a host file or another group file. During the search procedure all elements of a group file are checked; however they are inspected in random order.

The host file contains all the useful information that the client needs: the interface id, interface version and one or more of: protocol sequence, network address and end-point information.

4.2 NIS+NSI DATA STRUCTURE

Three different data structure are used in NIS+NSI: BIND_DATA, SERVER_ENTRY and RPC_NS_HANDLE.

The server's binding information is kept in BIND_DATA structure, which has the following format:

```
typedef struct binding_data{
    char    prot_seq[4];           /* protocol sequence */
    char    net_addr[MAX_ADDR]; /* network address */
    int     end_pt;               /* server's end point */
    char    host[N_MAX];          /* server's name */
} BIND_DATA;
```

The information kept in each server entry is represented in the SERVER_ENTRY data structure:

```
typedef struct server_entry_cont{
    unsigned long   face_id;      /* RPC interface id */
    int             face_ver;     /* RPC interface version */
    BIND_DATA      bind_array[B_MAX]; /*array of binding info. */
} SERVER_ENTRY;
```

The last data structure holds the context information:

```
typedef struct handle{
    unsigned long   if_id;        /* RPC interface id */
    int             if_ver;       /* RPC interface version */
}
```

```

        int          max_bind;      /* max. number of bindings */
        char         name[N_MAX];
    } RPC_NS_HANDLE;

```

Here MAX_ADDR = 16, N_MAX = 50 and B_MAX = 10.

4.3 NIS+NSI NAMING CRITERIA

File names must have the following form:

prof/grp/host_ service_ [any other characteristic]_version number

4.4 NIS+NSI LIBRARY ROUTINES

There are four imperative and six auxiliary routines that make up the NIS+NSI interface. The four imperative routines are: `ns_export()`, `ns_import()`, `ns_lookup()` and `ns_qimport()`. The auxiliary routines are: `ns_import_begin()`, `ns_import_done()`, `ns_lookup_begin()`, `ns_lookup_done`, `ns_qimport_begin()`, `ns_qimport_done()`.

*ns_export(char * server_name, SERVER_ENTRY server_info)*

This is the only interface used by servers. Servers call `ns_export()` to advertise RPC interfaces. The `server_name` argument is the name of the file created by NIS+NSI, and must comply with the naming standard.

*ns_import(BIND_DATA **bind_ptr, RPC_NS_HANDLE *import_context)*

Clients call this routine to get one binding information for the requested interface id and version. Clients must first call `ns_import_begin()` to get a context, which is `import_context`, and pass it to `ns_import()`.

*ns_lookup(BIND_DATA return_vector[], RPC_NS_HANDLE *lookup_context)*

Clients call this routine to get an array of binding information for the requested interface id and version. Once again, clients must first call `ns_lookup_begin()` to get `lookup_context`, so as to be able to pass it to `ns_lookup()` function.

*ns_qimport(BIND_DATA **bind_ptr, RPC_NS_HANDLE *qimport_context, svp_nodestates_c_t states)*

To get binding information for the best available workstation, clients can call this routine, which takes advantage of facilities provided by Quahog, a distributed system developed at Brown. This routine also requires the clients to call `ns_qimport_begin()` to get a `qimport_context` and pass it as argument.

ns_import_begin(char name[], unsigned long id, int version, RPC_NS_HANDLE

*****import_context)***

ns_lookup_begin(char name[], unsigned long id, int version, RPC_NS_HANDLE

*****lookup_context)***

ns_qimport_begin(char name[], unsigned long id, int version, RPC_NS_HANDLE

*****qimport_context)***

These routines create a context for client's specific request, using the information given as argument. The first argument, *name*, is the name of the file at which the search should be started.

ns_import_done(RPC_NS_HANDLE **import_context)

ns_lookup_done(RPC_NS_HANDLE **lookup_context)

ns_qimport_done(RPC_NS_HANDLE **qimport_context)

These routines destroy the context created by the various*_begin()* routines. Clients call one of these routines to destroy the context after the RPC call is over. However, if the client wants to make more calls to the same host for the same RPC interface later on, then she/he can save the context and call ...*_done()* routine after all calls have been made.

4.5 DATABASE SETUP

In the previous section, *ns_export()* was introduced as the interface for creating host files. Host files can be created easily by the server. But in order to perform different searches, the host entries must be placed in the leaf level of the search tree. Profile and group files form the internal nodes and the root of the search tree.

As mentioned in Section 4.1, profile and group files are created externally. It is the system administrator's responsibility to create profile and group files. Although it is not a hard task, the file creation must comply with the order and specifications noted in Section 4.1. Otherwise, NIS+NSI features will not be useful. A specific directory must be made to hold profile and group files. The profile and group files are currently kept in */u/shm/nis* directory.

5. MATRIX MULTIPLICATON, A PARALLEL APPLICATION

Matrix multiplication was chosen as an example to illustrate the use and effectiveness of NIS+NSI. This application takes advantage of the *ns_lookup()* family of interfaces as well as the Solaris thread package and Sun's RPC to compute the multiplication of rows by columns of matrices in parallel. The application consists of the client-side program, server-side program, and the protocol definition file. Since

the application uses SUN's protocol compiler (RPCGEN); there is also an XDR wrapper routine, client and server stubs and finally a common .h file. A short description of the server-side program and the protocol definition file is followed; however, the focus of this section is on the client-side program, which utilizes NIS+NSI. The server-side routine performs vector multiplication: given two vectors, namely the row vector of matrix A and the column vector of matrix B, this routine returns the resultant matrix C component. The protocol definition file defines a vector and the program definition which is required by SUN RPC.

In this specific example, because of the nature of the computation and in order to take full advantage of the Solaris thread package, the client needs to get an array of hosts so that it can send different vector multiplications to different hosts for computation. Therefore the *ns_lookup()* routine and its auxiliary routines are called to provide a list of hosts. However, if the client wants to create some number of threads but send them to only one host for a concurrent computation, then other interfaces like *ns_import()* or *ns_qimport()* could be used. Certainly, *ns_qimport()* would have been a better choice, because of its ability to find the most idle workstation to do the job faster. It is important to note that the application programmer is responsible for memory allocation for different contexts that are created by the various *..._begin()* routines.

Four hosts have been exported for the matrix example; three of them have the same RPC interface ID and version number, and the fourth has a different version number. Therefore, two group files are created to represent different groups of hosts. The input file contains the default profile name as the starting point for search. The profile file contains the name of both groups. The search path starts from the default file and follows to the group file, which leads to the hosts providing the desired RPC interface ID and version number and ends at the host level.

Note that in this example, six threads are created to do the vector multiplication; however, only three hosts are returned from *ns_lookup()* upon the client's request. The use of producer/consumer problem is necessary in order to safely distribute hosts between threads. Upon getting the list of hosts from *ns_lookup()*, host names are added (producer) to a buffer. Before creating a client, a host name is removed (consumer) from the buffer. Synchronization is done using condition variables to represent the queue of the threads waiting for getting a host name or putting the host name back after it is done.

The protocol definition file (*matrix.x*), server-side program (*matrix_proc.c*), client-side program (*matrix.c*) and the output of the computation are included here for reference.

Bibliography

1- Solaris 2.3 System Administrator AnswerBook

2- OSF DCE Application Development Guide

```
/******  
/*                               matrix.x                               */  
/*                               */  
/* This is the protocol definition file for the matrix multiplication */  
/* RPC service.                                                         */  
/******  
  
const MAX_ROW_COL = 20; /* maximum number of rows or columns for each matrix*/  
const MAX_THR = 50;    /* maximum number of threads */  
  
typedef int vect<MAX_ROW_COL>; /* array decralation for XDR */  
  
/* program definition */  
  
program MATRXPROG {  
    version MATRXVERS {  
        int MATRIX(vect,vect) = 1;  
    } = 1;  
} = 0x27000000;
```

```

/*****
/*
/*          matrix_proc.c
/*
/* This the server side of the matrix multiplication RPC service. It computes
/* the multiplication of one row of the first matrix by one column of second
/* matrix. It returns a boolean value showing if the computation was sucess-
/* ful or not. The inputs are: two vectors representing the row and the
/* of the matrices. The result of the computation will be placed in parameter
/* result, sent in the argument list.
/* Input: Two vectors representing the row and the columns of the matrices,
/* Output: A boolean value showing if the computation was sucessful or not
/*          the result of the computation is placed in result parameter.
/* Effect: Computes the value of the matrix C's component by multiplying two
/*          given vectors.
*****/

```

```

#include <stdio.h>
#include <rpc/rpc.h>
#include "matrix.h"

```

```

bool_t
matrix_l_svc(vect_a, vect_b, result, rqstp)
vect    vect_a;
vect    vect_b;
int     *result;
struct  svc_req *rqstp;
{
    int k;
    bool_t  ret_val;          /* boolean value indicating the sucessfulness
                             of computation */

    /* vect_a.vect_val and vect_b.vect_val are pointers to the arrays
       vect_a and vect_b, based on XDR representation. */
    *result = 0;
    for(k=0; k<vect_a.vect_len; k++)
    {
        *result += (*(vect_a.vect_val+k)) * (*(vect_b.vect_val+k));
    }
    ret_val = 1;
    return (ret_val);
}

```

```

int
matrxprog_l_freeresult(transp, xdr_result, result)
SVCXPRT *transp;
xdrproc_t xdr_result;
caddr_t result;
{
    (void) xdr_free(xdr_result, result);

    /*
     * Insert additional freeing code here, if needed
     */
    return(1);
}

```

```

/*****
/*
/*          matrix.c          */
/*
/* This is the client side of the matrix multiplication RPC service. It takes
/* advantage of solaris thread package, to create one thread for each multip-
/* lication of one row by one column. After filling the matrices with the
/* vlaues from an input file, it calls ns_lookup(), to get a list of servers.
/* Then creates the threads. Each thread calls a routine which in turn does
/* the RPC call. After a RPC call has been sucessfully done, the server's name
/* will be added to a buffer(it is produced). If for some reason the RPC call
/* was unsuccessfull, or the a connection was not established, then the thread
/* will join the queue for the consuming a server. This will gurantee that the
/* job will be done.
*****/

```

```

#include "matrix.h"
#include "matrix_client.h"
#include <nis.h>

```

```

/*
 * function declaration
 */

```

```

void next_step();
void call_rpc();
void producer();
char *consumer();
void print_matrix();
void set_arg_sent();
void call_producer();

```

```

/*
 * global variables
 */

```

```

buffer    *b;          /* buffer holding the servers's name */
int       num_servers;
int       A[MAX_ROW_COL][MAX_ROW_COL];
int       B[MAX_ROW_COL][MAX_ROW_COL];
int       C[MAX_ROW_COL][MAX_ROW_COL];

        /* the argument packed to be sent when creating the thread */
struct    argpack{
    int     current_row; /* the row number for the first matrix */
    int     current_col; /* the column number for the second matrix */
    vect    vect_a;      /* row vector */
    vect    vect_b;      /* column vector */
};

```

```

/*****
/*
/*          main()          */
/*
/* This initializes all global variables. Then calls next_step to create
/* the threads.
*****/

```

```

void main(argc, argv)
int  argc;

```



```

char *argv[];
{
    /* declare temporary variables */
    char    name[MAX_ROW_COL];
    int     max,i,j,k,n,m,a_col,a_row,b_col;
    FILE    *fp;

b = malloc(sizeof(struct buffer_t));
b->nextin = 0;
b->nextout = 0;
b->occupied = 0;

printf("\n\n                Matrix Multiplication\n");
printf("                Application Output\n\n\n");

                /* open input file */
if((fp = fopen(argv[1],"r")) == NULL)
{
    printf("can't open the %s file\n",argv[1]);
    exit(0);
}
else
{
                /* fill the matrices with the input values */
while( ! feof(fp))
{
    fscanf(fp,"%s\n", name);
    fscanf(fp,"%d %d %d %d", &max, &a_row, &a_col, &b_col);
    printf("\n\nMax. number of servers requested: %d\n",max);
    printf("Number of rows and columns in matrix A: %d %d\n",a_row,a_col);
    printf("Number of columns in matrix B: %d\n\n",b_col);
                /* get and init. the matrices information */
for(i=0; i<a_row; i++)
    for(j=0; j<a_col; j++){
        fscanf(fp,"%d", &A[i][j]);
        printf("A[%d][%d] = %d\n",i,j,A[i][j]);}
for(j=0; j<a_col; j++)
    for(k=0; k<b_col; k++){
        fscanf(fp, "%d", &B[j][k]);
        printf("B[%d][%d] = %d\n",j,k,B[j][k]);}

for(n=0; n<a_col; n++)
    for(m=0; m<b_col; m++)
        C[n][m] = 0;

}

    next_step(name,max,a_row,a_col,b_col);
}
fclose(fp);
}

/*****
/*
/*                next_step()
/*
/* This allocates memory for lookup_context, calls ns_lookup_begin() to
/* get a context and then calls ns_lookup() to get an array of hosts. It
/* first creates a thread which calls call_producer(), to put the hosts
/* returned from ns_lookup() in the buffer. Then it will create the threads*/
/* to call RPC and get the values of the matrix C. At the end calls print-*/
/* matrix to print the matrices.
/*
/*
/* Input: The name of the file to start the search, the max. number of ho_*
/* sts requested by the client, the number of rows and columns of
/* matrix A, and the number of columns of matrix B.
/*

```

```

/* Output: none */
/* Effect: creates some number of threads to do the computation in parallel */
/*****

void next_step(name,max,a_row,a_col,b_col)
char name[MAX_ROW_COL];
int max,a_row,a_col,b_col;
{
    int i,j,m;
    RPC_NS_HANDLE *lookup_context;
    BIND_DATA bind_vect[BIND_MAX]; /* array of binding information */
    struct argpack *arg_send; /* argument sent with creation of thread*/

    lookup_context = (RPC_NS_HANDLE *) (malloc (sizeof(struct handle)));

    ns_lookup_begin(name,MATRXPROG,MATRXVERS,max,&lookup_context);

    /* call ns_lookup() to get a list of servers */
    printf("\n\nCalling ns_lookup():\n");
    if ( ! (num_servers = ns_lookup(bind_vect,lookup_context)) )
        printf(" No binding info for this interface id \n");
    else
    {
        /* create an extra thread to balance the number of
           producing and consuming */
        if(thr_create(NULL, 0, (void *(*)(void *))call_producer,
            (void *) (bind_vect), 0, NULL) != 0)
        {
            perror("thr_create");
            exit(1);
        }

        for(i=0; i<a_row; i++)
            for(j=0; j<b_col; j++)
            {
                /* allocate memory for each thread's argument */
                arg_send = (struct argpack *) (malloc (sizeof(struct argpack)));
                set_arg_send(i,j,a_col,arg_send);

                /* create the threads */
                if(thr_create(NULL, 0, (void *(*)(void *))call_rpc,
                    (void *) (arg_send), 0, NULL) != 0)
                {
                    perror("thr_create");
                    exit(1);
                }
            }

        while(thr_join(0,0,0) == 0)
            ;
        print_matrix(a_row,a_col,b_col);
    }
    ns_lookup_done(&lookup_context);
    free(lookup_context);
}

/*****
/* call_producer() */
/* Routine which produces all servers returned from ns_lookup(), to be */
/* consumed by the RPC calls. */
/*****/

```

```

/* Input: the array of binding information */
/* Output: none */
/* Effect: calls producer() to put all hosts in the buffer, and then consu-*/
/*          mes some of them in order to balance the number of produces and */
/*          consumes. */
/*****

```

```

void call_producer(bind_vect)
BIND_DATA bind_vect[BIND_MAX];
{
    int i;

    for (i=0; i<num_servers; i++)
        {
            producer(bind_vect[i].host);
        }

    if(num_servers>BSIZE)
        for(i=0; i<num_servers-BSIZE;i++)
            consumer();
}

```

```

/*****
/*          set_arg_sent() */
/*          */
/* Routine to set the values in the argpack for each thread created. */
/*          */
/* Input: The row and column number of matrices A and B, number of elements*/
/*          in each vector, and the structure that holds these information. */
/* Output: The initialized structure */
/* Effect: initializes the structure with the given information. */
/*****

```

```

void set_arg_sent(i,j,num_elem,args)
int i,j;
int num_elem;
struct argpack *args;
{
    int m;
    int *temp_a; /* pointer to temporary vector */
    int *temp_b;

    temp_a = (int *) (malloc(num_elem * sizeof(int)));
    temp_b = (int *) (malloc(num_elem * sizeof(int)));

    args->current_row = i;
    args->current_col = j;
    args->vect_a.vect_len = num_elem;
    args->vect_b.vect_len = num_elem;

    for(m=0; m<num_elem; m++)
        {
            *(temp_a+m) = A[i][m];
            *(temp_b+m) = B[m][j];
        }

    args->vect_a.vect_val = temp_a;
    args->vect_b.vect_val = temp_b;
}

```

```

/*****
/*
/*          call_rpc()          */
/*
/* This is the threads' call back routine. It calls consumer() to get a
/* server. Then does a RPC call and places the result in matrix C.
/*
/*
/* Input: the structure args, which holds all information for doing RPC.
/* Output: none
/* Effect: initializes the matrix C by doing RPC call and getting values
/* from server routine.
*****/

```

```

void call_rpc(args)
struct argpack *args;
{
    CLIENT          *cl=NULL;    /* client handle */
    char            *host;       /* host's name */
    int             k,result;
    enum clnt_stat  ret_val;     /* boolean value representing the success of call*/

    /* while a host is got and the client is created correctly */
    while(cl == NULL)
    {
        host = consumer();
        cl = clnt_create(host, MATRXPROG, MATRXVERS, "netpath");
    }

    /* call server routine */
    ret_val = matrix_1(args->vect_a,args->vect_b,&result,cl);

    /* if the call was not successful */
    if (ret_val != RPC_SUCCESS)
    {
        clnt_perror(cl, "call failed");
    }

    /* assign the value to matrix C */
    C[args->current_row][args->current_col] = result;
    /* put the host back in the buffer for later use */
    producer(host);
    /* destroy client handle */
    clnt_destroy(cl);
}

```

```

/*****
/*
/*          producer()          */
/*
/* This tries to fill up the buffer with the host names, but it has to wait
/* until a slot in the buffer is available.
/*
/*
/* Input: the host name to be added to the buffer.
/* Output: none
/* Effect: fills up the buffer
*****/

```

```

void producer(item)
char *item;
{
    int i;

    if(mutex_lock(&b->mutex)<0)
    {
        perror("producer mutex_lock");
    }
}

```

```

    exit(1);
};

while(b->occupied >= BSIZE)
    if(cond_wait(&b->less, &b->mutex)<0)
    {
        perror("producer con_wait");
        exit(1);
    };

assert(b->occupied < BSIZE);
/* there is an empty slot available */
b->buf[b->nextin++] = item;
b->nextin %= BSIZE;
b->occupied++;

if(cond_signal(&b->more)<0)
{
    perror("producer cond_sign");
    exit(1);
};

if(mutex_unlock(&b->mutex)<0)
{
    perror("producer mutex_unlock");
    exit(1);
};
}

/*****
/*
/*          consumer()
/*
/* This takes the host names from the buffer, but it has to wait until a
/* host name is available in the buffer, then it returns the name to the
/* caller.
/*
/* Input: none
/* Output: a host name
/* Effect: removes a name from buffer, making the slot available
*****/

char* consumer()
{
char *item;
int i;

item = (char *) (malloc(sizeof(char)));

if(mutex_lock(&b->mutex)<0)
{
    perror("consumer mutex_lock");
    exit(1);
};
while(b->occupied <= 0)
    if(cond_wait(&b->more, &b->mutex)<0)
    {
        perror("consumer cond_wait");
        exit(1);
    };

assert(b->occupied > 0);
/* there is a host name available */

```

```

item = b->buf[b->nextout++];
b->nextout %= BSIZE;
b->occupied--;

if(cond_signal(&b->less)<0)
{
    perror("consumer cond_sign");
    exit(1);
};

if(mutex_unlock(&b->mutex)<0)
{
    perror("consumer mutex_unlock");
    exit(1);
};

return(item);
}

```

```

/*****
/*          print_matrix()          */
/*          */
/* This prints the matrices A, B and C.          */
/*          */
/* Input: Matrix A's number of row and columns and matrix B's number of          */
/*          columns.          */
/* Output: none          */
/* Effect: none          */
*****/

```

```

void print_matrix(a_row,a_col,b_col)
int a_row;
int a_col;
int b_col;
{
    int i,j,k,p;
printf("\n\n          PRINT MATRIX \n\n");
    for(i=0; i<(a_row + abs(a_row - a_col)); i++)
    {
        for(j=0; j<a_col; j++)
        {
            if(i < a_row)
            {
                print_line(j);
                printf("%2d ", A[i][j]);
                if(j == a_col-1) printf("|");
            }
            else
            {
                printf(" ");
                if(j == a_col-1) printf(" ");
            }
        }

        if(i == a_row/2) printf(" X ");

        for(k=0; k<b_col; k++)
        {
            if((i != a_row/2) && k == 0) printf(" ");
            if(i < a_col)
            {
                print_line(k);
            }
        }
    }
}

```

```

printf("%2d ", B[i][k]);
if(k == b_col-1) printf("|");

/* B's number of rows is greater than A's */
if((a_row < a_col) && (i >= a_row) && (k == b_col-1))
printf("\n");
}
else printf(" ");
}

/* B's number of rows is less than A's */
if((a_row > a_col) && (i >= a_col))
printf("%3d", ' ');

if(i == a_row/2) printf(" = ");

if(i < a_row)
for(p=0; p<b_col; p++)
{
if((i != a_row/2) && p == 0) printf(" ");
if(i < a_row)
{
print_line(p);
printf("%3d ", C[i][p]);
if(p == b_col-1) printf("|\\n");
}
else printf(" ");
}
}
}

```

Matrix Multiplication
Application Output

Max. number of servers requested: 3
Number of rows and columns in matrix A: 3 4
Number of columns in matrix B: 2

A[0][0] = 1
A[0][1] = 2
A[0][2] = 3
A[0][3] = 4
A[1][0] = 5
A[1][1] = 6
A[1][2] = 7
A[1][3] = 8
A[2][0] = 9
A[2][1] = 10
A[2][2] = 11
A[2][3] = 12
B[0][0] = 8
B[0][1] = 7
B[1][0] = 6
B[1][1] = 5
B[2][0] = 4
B[2][1] = 3
B[3][0] = 2
B[3][1] = 1

Calling ns_lookup():

reading profile: /u/shm/nis/prof_my_first
reading group file: /u/shm/nis/grp_matrix_1
reading host file: /u/shm/nis/host_matrix_3
reading host file: /u/shm/nis/host_matrix_2

PRINT MATRIX

1	2	3	4			8	7			40	30	
5	6	7	8	X	6	5	=	120	94			
9	10	11	12			4	3			200	158	
						2	1					