NIS+ Programming Guide 5

This chapter presents the fundamental principles of the NIS+ applications programming interface and a detailed sample program. The NIS+ API is for programmers who need to build applications for Solaris-based networks. It provides the essential features for supporting enterprise-wide applications.

NIS+ Overview	*page 149*
NIS+ API	*page 153*
NIS+ Sample Program	*page 157*

NIS+ Overview

The NIS+ network name service addresses the requirements of client/server networks ranging in size from 10 clients supported by a few servers on a simple local area network to 10,000 multi-vendor clients supported by 20 to 100 specialized servers located in sites throughout the world and connected by several public networks.

Domains

NIS+ supports hierarchical domains, illustrated as a simple case in Figure 5-1.

5

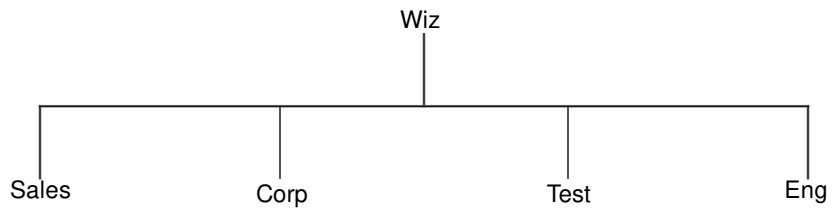

Figure 5-1 NIS+ Domain

A NIS+ domain is a set of data describing the workstations, users, and network services in a portion of an organization. NIS+ domains can be administered independently of each other. This allows NIS+ to be used in a range of networks, from small to very large.

Servers

Each domain is supported by a set of servers. The principal server is called the master server, and the backup servers are called replicas. Both master and replica servers run NIS+ server software. The master server stores the original tables, and the backup servers store copies.

NIS+ accepts incremental updates to the replicas. Changes are first made on the master server. Then they are automatically propagated to the replica servers and are soon available to the entire namespace.

Tables

NIS+ stores information in tables instead of maps or zone files. NIS+ provides 16 types of predefined, or system, tables, shown in Figure 5-2.

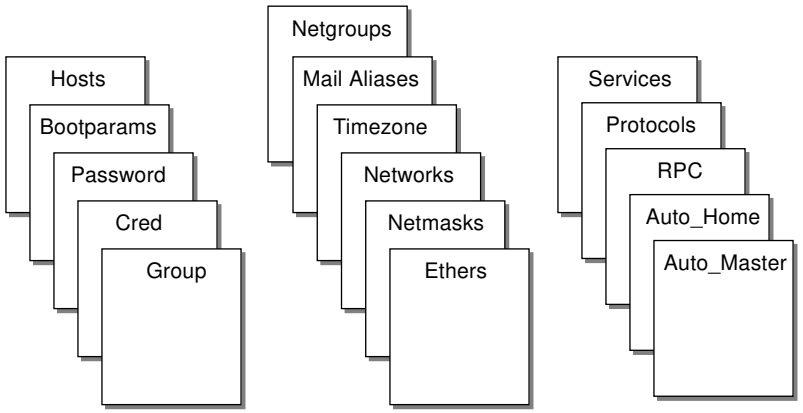

Figure 5-2 NIS+ Tables

Each table stores a different type of information. For instance, the Hosts table stores host name/Internet address pairs, and the Password table stores information about users of the network.

NIS+ tables have two major improvements over NIS maps. First, a NIS+ table can be accessed by any column, not just the first column (sometimes referred to as the "key"). This eliminates the need for duplicate maps, such as the `hosts.byname` and `hosts.byaddr` maps of NIS. Second, access to the information in NIS+ tables can be controlled at three levels of granularity: the table level, the entry level, and the column level.

NIS+ Security

The NIS+ security model provides both authorization and authentication mechanisms. First, every object in the namespace specifies the type of operation it accepts and from whom. This is authorization. Second, NIS+ attempts to authenticate every requestor accessing the namespace. Once it identifies the originator of the request, it determines whether the object has authorized that particular operation for that particular principal. Based on its authentication and the object's authorization, NIS+ carries out or denies the access request.

5

Name Service Switch

NIS+ works in conjunction with a separate facility called the Name Service Switch. The Name Service Switch, sometimes referred to as "the Switch," lets Solaris 2.x-based workstations obtain their information from more than one network information service; specifically, from local, or `/etc` files, from NIS maps, from DNS zone files, or from NIS+ tables. The Switch not only offers a choice of sources, but allows a workstation to specify different sources for different *types* of information. The name service is configured through the file `/etc/nsswitch.conf`().

NIS+ Administration Commands

NIS+ provides a full set of commands for administering a namespace. Table 5-1 summarizes them.

Table 5-1 NIS+ Namespace Administration Commands

Command	Description
`nischgrp`	Changes the group owner of a NIS+ object.
`nischmod`	Changes an object's access rights.
`nischown`	Changes the owner of a NIS+ object.
`nisgrpadm`	Creates or destroys a NIS+ group, or displays a list of its members. Also adds members to a group, removes them, or tests them for membership in the group.
`niscat`	Displays the contents of NIS+ tables.
`nisgrep`	Searches for entries in a NIS+ table.
`nisls`	Lists the contents of a NIS+ directory.
`nismatch`	Searches for entries in a NIS+ table.
`nisaddent`	Adds information from `/etc` files or NIS maps into NIS+ tables.
`nistbladm`	Creates or deletes NIS+ tables, and adds, modifies or deletes entries in a NIS+ table.
`nisaddcred`	Creates credentials for NIS+ principals and stores them in the Cred table.
`nispasswd`	Changes password information stored in the NIS+ Passwd table.
`nisupdkeys`	Updates the public keys stored in a NIS+ object.

Table 5-1 NIS+ Namespace Administration Commands

Command	Description
`nisinit`	Initializes a NIS+ client or server.
`nismkdir`	Creates a NIS+ directory and specifies its master and replica servers.
`nisrmdir`	Removes NIS+ directories and replicas from the namespace.
`nissetup`	Creates `org_dir` and `groups_dir` directories and a complete set of (unpopulated) NIS+ tables for a NIS+ domain.
`rpc.nisd`	The NIS+ server process.
`nis_cachemgr`	Starts the NIS+ Cache Manager on a NIS+ client.
`nischttl`	Changes a NIS+ object's time-to-live value.
`nisdefaults`	Lists a NIS+ object's default values: domain name, group name, workstation name, NIS+ principal name, access rights, directory search path, and time-to-live.
`nisln`	Creates a symbolic link between two NIS+ objects.
`nisrm`	Removes NIS+ objects (except directories) from the namespace.
`nisshowcache`	Lists the contents of the NIS+ shared cache maintained by the NIS+ Cache Manager.

NIS+ API

The NIS+ application programming interface (API) is a group of functions that can be called by an application to access and modify NIS+ objects. The NIS+ API has 54 functions that fall into nine categories:

- Object Manipulation Functions (`nis_names`)
- Table Access Functions (`nis_tables`)
- Local Name Functions (`nis_local_names`)
- Group Manipulation Functions (`nis_groups`)
- Server Related Functions (`nis_server`)
- Database Access Functions (`nis_db`)
- Error Message Display Functions (`nis_error`)
- Transaction Log Functions (`nis_admin`)
- Miscellaneous Functions (`nis_subr`)

The functions in each category are summarized in Table 5-2.
The category names match the names by which they are grouped in the NIS+ manpages.

Table 5-2 NIS+ API Functions

Function	Description
nis_names	**Locate and Manipulate Objects**
nis_lookup()	Returns a copy of an NIS+ object. Can follow links. Though it cannot search for an entry object, if a link points to one, it can return an entry object.
nis_add()	Adds an NIS+ object to the namespace.
nis_remove()	Removes an NIS+ object in the namespace.
nis_modify()	Modifies an NIS+ object in the namespace.
nis_tables	**Search and Update Tables**
nis_list()	Searches a table in the NIS+ namespace and returns entry objects that match the search criteria. Can follow links and search paths from one table to another.
nis_add_entry()	Adds an entry object to an NIS+ table. Can be instructed to either fail or overwrite if the entry object already exists. Can return a copy of the resulting object if the operation was successful.
nis_freeresult()	Frees all memory associated with a nis_result structure.
nis_remove_entry()	Removes one or more entry objects from an NIS+ table. Can identify the object to be removed by using search criteria or by pointing to a cached copy of the object. If using search criteria, can remove all objects that match the search criteria; therefore, with the proper search criteria, can remove all entries in a table. Can return a copy of the resulting object if the operation was successful.
nis_modify_entry()	Modifies one or more entry objects in an NIS+ table. Can identify the object to be modified by using search criteria or by pointing to a cached copy of the object.
nis_first_entry()	Returns a copy of the first entry object in an NIS+ table.

Table 5-2 NIS+ API Functions

Function	Description
`nis_next_entry()`	Returns a copy of the "next" entry object in an NIS+ table. Because a table can be updated and entries removed or modified between calls to this function, the order of entries returned may not match the actual order of entries in the table.
`nis_local_names`	**Get Default Names for the Current Process**
`nis_local_directory()`	Returns the name of the workstation's NIS+ domain.
`nis_local_host()`	Returns the fully-qualified name of the workstation. A fully qualified name has the form <host-name>.<domain-name>.
`nis_local_group()`	Returns the name of the current NIS+ group, which is specified by the environment variable NIS_GROUP.
`nis_local_principal()`	Returns the name of the NIS+ principal whose UID is associated with the calling process.
`nis_getnames()`	Returns a list of possible expansions to a particular name.
`nis_freenames()`	Frees the memory containing the list generated by `nis_getnames`.
`nis_groups`	**Group Manipulation and Authorization**
`nis_ismember()`	Test whether a principal is a member of a group.
`nis_addmember()`	Adds a member to a group. The member can be a principal, a group, or a domain.
`nis_removemember()`	Deletes a member from a group.
`nis_creategroup()`	Create a group object.
`nis_destroygroup()`	Delete a group object.
`nis_verifygroup()`	Tests whether a group object exists.
`nis_print_group_entry()`	Lists the principals that are members of a group object.
`nis_server`	**Various services for NIS+ applications.**
`nis_mkdir()`	Creates the databases to support service for a named directory on a specified host.
`nis_rmdir()`	Removes the directory from a host.

5

Table 5-2 NIS+ API Functions

Function	Description
`nis_servstate()`	Sets and reads state variables of NIS+ servers and flushes internal cashes.
`nis_stats()`	Retrieves statistics about a server's performance.
`nis_getservlist()`	Returns a list of servers that support a particular domain.
`nis_freeservlist()`	Frees the list of servers returned by `nis_getservlist`.
`nis_freetags()`	Frees the memory associated with the results of `nis_servstate` and `nis_stats`.
`nis_db`	**Interface Between the NIS+ Server and the Database. Not To Be Used By a NIS+ Client.**
`db_first_entry()`	Returns a copy of the first entry of the specified table.
`db_next_entry()`	Returns a copy of the entry succeeding the specified entry.
`db_reset_next_entry()`	Terminates a first/next entry sequence.
`db_list_entries()`	Returns copies of entries that meet specified attributes.
`db_remove_entry()`	Removes all entries that meet specified attributes.
`db_add_entry()`	Replaces an entry in a table identified by specified attributes with a copy of the specified object, or adds the object to the table.
`db_checkpoint()`	Reorganizes the contents of a table to make access to the table more efficient.
`db_standby()`	Advises the database manager to release resources.
`nis_error`	**Functions that supply descriptive strings equivalent to NIS+ status values**
`nis_sperrno()`	Returns a pointer to the appropriate string constant.
`nis_perror()`	Displays the appropriate string constant on standard output.
`nis_lerror()`	Sends the appropriate string constant to syslog
`nis_sperror()`	Returns a pointer to a statically allocated string to be used or to be copied with `strdup`().
`nis_admin`	**Transaction logging functions used by servers**
`nis_ping()`	Used by the master server of a directory to time stamp it. This forces replicas of the directory to be updated.

Table 5-2 NIS+ API Functions

Function	Description
`nis_checkpoint()`	Forces logged data to be stored in the table on disk.
`nis_subr`	**Functions To Help Operate on NIS+ Names and Objects.**
`nis_leaf_of()`	Returns the first label in an NIS+ name. The returned name does not have a trailing dot.
`nis_name_of()`	Removes all domain-related labels and returns only the unique object portion of the name. The name passed to the function must be either in the local domain or in one of its child domains, or the function returns NULL.
`nis_domain_of()`	Returns the name of the domain in which an object resides. The returned name ends in a dot.
`nis_dir_cmp()`	Compares any two NIS+ names. The comparison ignores case and states whether the names are the same, descendants of each other, or not related.
`nis_clone_object()`	Creates an exact duplicate of an NIS+ object.
`nis_destroy_object()`	Destroys an object created by `nis_clone_object`.
`nis_print_object()`	Prints the contents of an NIS+ object structure to `stdout`.

NIS+ Sample Program

This program performs the following tasks:

- Determines the local principal and local domain
- Looks up the local directory object
- Creates a directory called `foo` under the local domain
- Creates the `groups_dir` and `org_dir` directories under domain foo
- Creates a group object `admins.foo`
- Adds the local principal to the admins group
- Creates a table under `org_dir.foo`
- Adds two entries to the `org_dir.foo` table
- Retrieves and displays the new membership list of the `admins` group
- Lists the namespace under the `foo` domain using callbacks
- Lists the contents of the table created using callbacks
- Cleans up all the objects that were created by removing the following:
  - the local principal from the admins group
  - the `admins` group

- the entries in the table followed by the table
- the `groups_dir` and `org_dir` directory objects
- the `foo` directory object

The example program is not a typical application. In a normal situation the directories and tables would be created or removed through the command line interface, and applications would manipulate NIS+ entry objects.

Unsupported Macros

The sample program uses unsupported macros that are defined in the file `<rpcsvc/nis.h>`. These are not public APIs and can change or disappear in the future. They are used for illustration purposes only and if you choose to use them, you do so at your own risk. The macros used are:

- `NIS_RES_OBJECT`
- `ENTRY_VAL`
- `DEFAULT_RIGHTS`

Functions Used in the Example

The use of the following NIS+ C API functions is illustrated through this example:

```
nis_add()               nis_add_entry()         nis_addmember()
nis_creategroup()       nis_destroygroup()      nis_domain_of()
nis_freeresult()        nis_leaf_of ()          nis_list()
nis_local_directory()   nis_local_principal()   nis_lookup()
nis_mkdir()             nis_perror()            nis_remove()
nis_remove_entry()      nis_removemember()
```

Program Compilation

The program shown in Code Example 5-1 assumes that the NIS+ principal running this application has permission to create directory objects in the local domain. The program is compiled:

```
yourhost% cc -o example.c example -lnsl
```

It is invoked:

```
yourhost% example [dir]
```

where dir is the NIS+ directory in which the program creates all the NIS+ objects. Specifying no directory argument causes the objects to be created in the parent directory of the local domain. Note that for the call to `nis_lookup()`, a space and the name of the local domain are appended to the string that names the directory. The argument is the name of the NIS+ directory in which to create the NIS+ objects. The principal running this program should have create permission in the directory.

Code Example 5-1 NIS+ Program Main `example.c`

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <rpcsvc/nis.h>

#define    MAX_MESG_SIZE 512
#define    BUFFER_SIZE 64
#define    TABLE_TYPE "test_data"

main(argc, argv)
    int     argc;
    char    *argv[];
{
    char    *saved_grp, *saved_name, *saved_owner;
    char    dir_name[NIS_MAXNAMELEN];
    char    local_domain[NIS_MAXNAMELEN];
    char    local_princip [NIS_MAXNAMELEN];
    char    org_dir_name [NIS_MAXNAMELEN];
    char    grp_name [NIS_MAXNAMELEN];
    char    grp_dir_name [NIS_MAXNAMELEN];
    char    table_name [NIS_MAXNAMELEN];
    nis_object     *dirobj, entdata;
    nis_result     *pres;
    u_int   saved_num_servers;
    int     err;

    if (argc == 2)
        sprintf (local_domain, "%s.", argv[1]);
    else
        strcpy (local_domain, "");

    strcat (local_domain, (char *) nis_local_directory());
    strcpy (local_princip, (char *) nis_local_principal());
    /*
     * Lookup the directory object for the local domain for two reasons:
```

```
 * 1.To get a template of a nis_object.
 * 2.To reuse some of the information contained in the directory
 * object returned. We could have declared a static nis_object, but
 * since we need to change very little, it is easier to make the
 * changes and not initialize the nis_object structure.
 */

pres = nis_lookup (local_domain, 0);
if (pres->status != NIS_SUCCESS) {
    nis_perror (pres->status, "unable to lookup local directory");
    exit (1);
}

/*
 * re-use most of the fields in the parent directory object - save
 * pointers to the fields that are being changed so that we can
 * free the original object and avoid dangling pointer references.
 */
dirobj = NIS_RES_OBJECT (pres);
saved_name = dirobj->DI_data.do_name;
saved_owner = dirobj->zo_owner;
saved_grp = dirobj->zo_group;

/*
 * set the new name, group, owner and new access rights for the
 * foo domain.
 */
sprintf (dir_name, "%s.%s", "foo", local_domain);
sprintf (grp_name, "%s.%s", "admins", dir_name);
dirobj->DI_data.do_name = dir_name;
dirobj->zo_group = grp_name;
dirobj->zo_owner = local_princip;

/*
 * Access rights in NIS+ are stored in a u_long with the highest
 * order bytes reserved for the "nobody" category, the next eight
 * bytes reserved for the owner, followed by group and world. In
 * this example we are giving access to the directory based on the
 * "----rmcdrmcd----" access right pattern.
 */
dirobj->zo_access = ((NIS_READ_ACC + NIS_MODIFY_ACC
                    + NIS_CREATE_ACC + NIS_DESTROY_ACC) << 16)
                    | ((NIS_READ_ACC + NIS_MODIFY_ACC
                    + NIS_CREATE_ACC + NIS_DESTROY_ACC) << 8);

/*
```

```
 * Save the number of servers the parent directory object had so
 * that we can restore this value before calling nis_freeresult()
 * later and avoid memory leaks.
 */
saved_num_servers = dirobj->DI_data.do_servers.do_servers_len;

/* We want only one server to serve this directory */
dirobj->DI_data.do_servers.do_servers_len = 1;

dir_create (dir_name, dirobj);

/* create the groups_dir and org_dir directories under foo. */
sprintf (grp_dir_name, "groups_dir.%s", dir_name);
dirobj->DI_data.do_name = grp_dir_name;
dir_create (grp_dir_name, dirobj);

sprintf (org_dir_name, "org_dir.%s", dir_name);
dirobj->DI_data.do_name = org_dir_name;
dir_create (org_dir_name, dirobj);

grp_create (grp_name);

printf ("\nAdding principal %s to group %s ... \n",
           local_princip, grp_name);
err = nis_addmember (local_princip, grp_name);
if (err != NIS_SUCCESS) {
    nis_perror (err,
        "unable to add local principal to group.");
    exit (1);
}

sprintf (table_name, "test_table.org_dir.%s", dir_name);
tbl_create (dirobj, table_name);

/*
 * Now create NIS+ entry objects in the table that was just created
 */
stuff_table (table_name);

/* Display what we stuffed */
list_objs(dir_name, table_name, grp_name);

/* Clean out everything we created. */
cleanup (local_princip, grp_name, table_name, dir_name, dirobj);

/*
```

```
     * Restore the saved pointers from the original pres structure
     * so that we can free up the associated memory and have no
     * memory leaks.
     */
    dirobj->DI_data.do_name = saved_name;
    dirobj->zo_group = saved_grp;
    dirobj->zo_owner = saved_owner;
    dirobj->DI_data.do_servers.do_servers_len = saved_num_servers;
    (void) nis_freeresult (pres);
}
```

Code Example 5-2 shows the routine is called by `main()` to create directory objects.

Code Example 5-2 NIS+ Routine to Create Directory Objects

```
void
dir_create (dir_name, dirobj)
    nis_name   dir_name;
    nis_object *dirobj;
{
    nis_result *cres;
    nis_error  err;

    printf ("\n Adding Directory %s to namespace ... \n", dir_name);
    cres = nis_add (dir_name, dirobj);
    if (cres->status != NIS_SUCCESS) {
        nis_perror (cres->status, "unable to add directory foo.");
        exit (1);
    }

    (void) nis_freeresult (cres);

    /*
     * NOTE: you need to do a nis_mkdir to create the table to store the
     * contents of the directory you are creating.
     */
    err = nis_mkdir (dir_name,
                     dirobj->DI_data.do_servers.do_servers_val);
    if (err != NIS_SUCCESS) {
        (void) nis_remove (dir_name, 0);
        nis_perror (err,
                "unable to create table for  directory object foo.");
        exit (1);
    }
}
```

This routine is called by `main()` to create the group object. Since `nis_creategroup`() works only on group objects, the "groups_dir" literal is not needed in the group name.

Code Example 5-3 NIS+ Routine to Create Group Objects

```
void
grp_create (grp_name)
    nis_name    grp_name;
{
    nis_error   err;

    printf ("\n Adding %s group to namespace ... \n", grp_name);
    err = nis_creategroup (grp_name, 0);
    if (err != NIS_SUCCESS) {
        nis_perror (err, "unable to create group.");
        exit (1);
    }
}
```

The routine shown in Code Example 5-4 is called by `main()` to create a table object laid out as shown in Table 5-3.

Table 5-3 NIS+ Table Objects

	Column1	**Column2**
Name:	id	name
Attributes:	Searchable, Text	Searchable, Text
Access Rights	----rmcdr---r---	----rmcdr---r---

The `TA_SEARCHABLE` constant indicates to the service that the column is searchable. Only `TEXT` (the default) columns are searchable. `TA_CASE` indicates to the service that the column value is to be treated in a case-insensitive manner during searches.

Code Example 5-4 NIS+ Routine to Create Table Objects

```
#define     TABLE_MAXCOLS 2
#define     TABLE_COLSEP ':'
#define     TABLE_PATH 0

void
tbl_create (dirobj, table_name)
    nis_object *dirobj;    /* need to use some of the fields */
```

```
    nis_name    table_name;
{
    nis_result      *cres;
    static nis_object  tblobj;
    static table_col   tbl_cols[TABLE_MAXCOLS] = {
        {"Id", TA_SEARCHABLE | TA_CASE, DEFAULT_RIGHTS},
        {"Name", TA_SEARCHABLE | TA_CASE, DEFAULT_RIGHTS}
    };

    tblobj.zo_owner = dirobj->zo_owner;
    tblobj.zo_group = dirobj->zo_group;
    tblobj.zo_access = DEFAULT_RIGHTS;   /* macro defined in nis.h  */
    tblobj.zo_data.zo_type = TABLE_OBJ;   /* enumerated type in nis.h
*/
    tblobj.TA_data.ta_type = TABLE_TYPE;
    tblobj.TA_data.ta_maxcol = TABLE_MAXCOLS;
    tblobj.TA_data.ta_sep = TABLE_COLSEP;
    tblobj.TA_data.ta_path = TABLE_PATH;
    tblobj.TA_data.ta_cols.ta_cols_len =
        tblobj.TA_data.ta_maxcol;       /* ALWAYS ! */
    tblobj.TA_data.ta_cols.ta_cols_val = tbl_cols;

/*
 * Use a fully qualified table name i.e. the "org_dir" literal should
 * be embedded in the table name. This is necessary because nis_add
 * operates on all types of NIS+ objects and needs the full path name
 * if a table is created.
 */
    printf ("\n Creating table %s ... \n", table_name);
    cres = nis_add (table_name, &tblobj);
    if (cres->status != NIS_SUCCESS) {
        nis_perror (cres->status, "unable to add table.");
        exit (1);
    }
    (void) nis_freeresult (cres);
}
```

The routine shown in Code Example 5-5 is called by main() to add entry objects to the table object. Two entries are added to the table object. Note that the column width in both entries is set to include the NULL character for a string terminator.

Code Example 5-5 NIS+ Routine to Add Objects to Table

```
#define    MAXENTRIES 2
void
stuff_table(table_name)
    nis_name table_name;
{
    int        i;
    nis_object entdata;
    nis_result *cres;
    static entry_col ent_col_data[MAXENTRIES][TABLE_MAXCOLS] = {
           {0, 2, "1", 0, 5, "John"},
           {0, 2, "2", 0, 5, "Mary"}
    };

    printf ("\n Adding entries to table ... \n");

    /*
     * Look up the table object first since the entries being added
     * should have the same owner, group owner and access rights as
     * the table they go in.
     */
    cres = nis_lookup (table_name, 0);
    if (cres->status != NIS_SUCCESS) {
        nis_perror (cres->status, "Unable to lookup table");
        exit(1);
    }
    entdata.zo_owner = NIS_RES_OBJECT (cres)->zo_owner;
    entdata.zo_group = NIS_RES_OBJECT (cres)->zo_group;
    entdata.zo_access = NIS_RES_OBJECT (cres)->zo_access;

    /* Free cres, so that it can be reused. */
    (void) nis_freeresult (cres);

    entdata.zo_data.zo_type = ENTRY_OBJ; /* enumerated type in nis.h
*/
    entdata.EN_data.en_type = TABLE_TYPE;
    entdata.EN_data.en_cols.en_cols_len = TABLE_MAXCOLS;
    for (i = 0; i < MAXENTRIES; ++i) {
        entdata.EN_data.en_cols.en_cols_val = &ent_col_data[i][0];
        cres = nis_add_entry (table_name, &entdata, 0);
        if (cres->status != NIS_SUCCESS) {
            nis_perror (cres->status, "unable to add entry.");
            exit (1);
        }
        (void) nis_freeresult (cres);
    }
}
```

The routine shown in Code Example 5-6 is the print function for the `nis_list()` call. When `list_objs()` calls `nis_list()`, a pointer to `print_info()` is one of the calling arguments. Each time the service calls this function, it prints the contents of the entry object. The return value indicates to the library to call with the next entry from the table.

Code Example 5-6 NIS+ Routine for `nis_list` Call

```
int
print_info (name, entry, cbdata)
    nis_name    name;       /* Unused */
    nis_object  *entry;     /* The NIS+ entry object */
    void        *cbdata;    /* Unused */
{
    static u_int    firsttime = 1;
    entry_col       *tmp;   /* only to make source more readable */
    u_int           i, terminal;

    if (firsttime) {
        printf ("\tId.\t\t\tName\n");
        printf ("\t---\t\t\t----\n");
        firsttime = 0;
    }
    for (i = 0; i < entry->EN_data.en_cols.en_cols_len; ++i) {
        tmp = &entry->EN_data.en_cols.en_cols_val[i];
        terminal = tmp->ec_value.ec_value_len;
        tmp->ec_value.ec_value_val[terminal] = '\0';
    }

    /*
     * ENTRY_VAL is a macro that returns the value of a specific
     * column value of a specified entry.
     */
    printf("\t%s\t\t\t%s\n", ENTRY_VAL (entry, 0),
                             ENTRY_VAL (entry, 1));
    return (0); /* always ask for more */
}
```

The routine shown in Code Example 5-7 is called by `main()` to list the contents of the group, table and directory objects. The routine demonstrates the use of callbacks also. It retrieves and displays the membership of the group. The group membership list is not stored as the contents of the object. So, it is

queried through the `nis_lookup()` instead of the `nis_list`() call. You must use the "groups_dir" form of the group name since `nis_lookup`() works on all types of NIS+ objects.

Code Example 5-7 NIS+ Routine to List Objects

```
void
list_objs(dir_name, table_name, grp_name)
         nis_name   dir_name, table_name, grp_name;
{
    group_obj  *tmp; /* only to make source more readable */
    u_int      i;
    char       grp_obj_name [NIS_MAXNAMELEN];
    nis_result *cres;
    char       index_name [BUFFER_SIZE];

    sprintf (grp_obj_name, "%s.groups_dir.%s",
               nis_leaf_of (grp_name), nis_domain_of (grp_name));
    printf ("\nGroup %s membership is: \n", grp_name);

    cres = nis_lookup(grp_obj_name, 0);
    if (cres->status != NIS_SUCCESS) {
        nis_perror (cres->status, "Unable to lookup group object.");
        exit(1);
    }
    tmp = &(NIS_RES_OBJECT(cres)->GR_data);
    for (i = 0; i < tmp->gr_members.gr_members_len; ++i)
        printf ("\t %s\n", tmp->gr_members.gr_members_val[i]);
    (void) nis_freeresult (cres);

    /*
     * Display the contents of the foo domain without using callbacks.
     */
    printf ("\nContents of Directory %s are: \n", dir_name);
    cres = nis_list (dir_name, 0, 0, 0);
    if (cres->status != NIS_SUCCESS) {
        nis_perror (cres->status,
                        "Unable to list Contents of Directory foo.");
        exit(1);
    }
    for (i = 0; i < NIS_RES_NUMOBJ(cres); ++i)
        printf("\t%s\n", NIS_RES_OBJECT(cres)[i].zo_name);
    (void) nis_freeresult (cres);

    /*
```

```
     * List the contents of the table we created using the callback form
     * of nis_list().
     */
    printf ("\n Contents of Table %s are: \n", table_name);
    cres = nis_list (table_name, 0, print_info, 0);
    if(cres->status != NIS_CBRESULTS && cres->status !=
NIS_NOTFOUND){
        nis_perror (cres->status,
            "Listing entries using callback failed");
        exit(1);
    }
    (void) nis_freeresult (cres);

    /*
     * List only one entry from the table we created. We will
     * use indexed names to do this retrieval.
     */

    printf("\n Entry corresponding to id 1 is:\n");
    /*
     * The name of the column is usually extracted from the table
     * object, which would have to be retrieved first.
     */
    sprintf(index_name, "[Id=1],%s", table_name);
    cres = nis_list (index_name, 0, print_info, 0);
    if(cres->status != NIS_CBRESULTS && cres->status !=
NIS_NOTFOUND){
        nis_perror (cres->status,
        "Listing entry using indexed names and callback failed");
        exit(1);
    }
    (void) nis_freeresult (cres);
}
```

The routine in Code Example 5-8 is called by `cleanup()` to remove a directory object from the namespace. It also informs the servers serving the directory about this deletion. Notice that the memory containing result structure, pointed to by `cres`, must be freed after the result has been tested.

Code Example 5-8 NIS+ Routine to Remove Directory Objects

```
void
dir_remove(dir_name, srv_list, numservers)
    nis_name    dir_name;
    nis_server *srv_list;
    u_int       numservers;
```

```
{
    nis_result *cres;
    nis_error  err;
    u_int      i;

    printf ("\nRemoving %s directory object from namespace ... \n",
                                dir_name);
    cres = nis_remove (dir_name, 0);
    if (cres->status != NIS_SUCCESS) {
        nis_perror (cres->status, "unable to remove directory");
        exit (1);
    }
    (void) nis_freeresult (cres);

    for (i = 0; i < numservers; ++i) {
        err = nis_rmdir (dir_name, &srv_list[i]);
        if (err != NIS_SUCCESS) {
            nis_perror (err,
            "unable to remove server from directory");
            exit (1);
        }
    }
}
```

This routine, Code Example 5-9, is called by `main()` to delete all the objects that were created in this example. Note the use of the `REM_MULTIPLE` flag in the call to `nis_remove_entry()`. All entries must be deleted from a table before the table itself can be deleted.

Code Example 5-9 NIS+ Routine to Remove All Objects

```
void
cleanup(local_princip, grp_name, table_name, dir_name, dirobj)
    nis_name   local_princip, grp_name, table_name, dir_name;
    nis_object *dirobj;
{
    char    grp_dir_name [NIS_MAXNAMELEN];
    char    org_dir_name [NIS_MAXNAMELEN];
    nis_error  err;
    nis_result *cres;

    sprintf(grp_dir_name, "%s.%s", "groups_dir", dir_name);
    sprintf(org_dir_name, "%s.%s", "org_dir", dir_name);

    printf("\n\n\nStarting to Clean up ... \n");
    printf("\n\nRemoving principal %s from group %s \n",
```

```
                    local_princip, grp_name);
err = nis_removemember (local_princip, grp_name);
if (err != NIS_SUCCESS) {
    nis_perror (err,
    "unable to delete local principal from group.");
    exit (1);
}

/*
 * Delete the admins group. We do not use the "groups_dir" form
 * of the group name since this API is applicable to groups only.
 * It automatically embeds the groups_dir literal in the name of
 * the group.
 */
printf("\nRemoving %s group from namespace ... \n", grp_name);
err = nis_destroygroup (grp_name);
if (err != NIS_SUCCESS) {
    nis_perror (err, "unable to delete group.");
    exit (1);
}

printf("\n Deleting all entries from table %s ... \n", table_name);
cres = nis_remove_entry(table_name, 0, REM_MULTIPLE);
switch (cres->status) {
    case NIS_SUCCESS:
    case NIS_NOTFOUND:
        break;
    default:
        nis_perror(cres->status, "Could not delete entries from
                                  table");
        exit(1);
}
(void) nis_freeresult (cres);

printf("\n Deleting table %s itself ... \n", table_name);
cres = nis_remove(table_name, 0);
if (cres->status != NIS_SUCCESS) {
    nis_perror(cres->status, "Could not delete table.");
exit(1);
}
(void) nis_freeresult (cres);

/* delete the groups_dir, org_dir and foo  directory objects. */
dir_remove (grp_dir_name,
                    dirobj->DI_data.do_servers.do_servers_val,
                    dirobj->DI_data.do_servers.do_servers_len);
```

```
    dir_remove (org_dir_name,
                      dirobj->DI_data.do_servers.do_servers_val,
                      dirobj->DI_data.do_servers.do_servers_len);
    dir_remove (dir_name, dirobj-
>DI_data.do_servers.do_servers_val,
                       dirobj->DI_data.do_servers.do_servers_len);
}
```

Running the program displays on the screen, as shown in Figure 5-3.

```
myhost% domainname
sun.com
myhost% ./sample
Adding Directory foo.sun.com. to namespace ...
Adding Directory groups_dir.foo.sun.com. to namespace ...
Adding Directory org_dir.foo.sun.com. to namespace ...
Adding admins.foo.sun.com. group to namespace ...
Adding principal myhost.sun.com. to group admins.foo.sun.com. ...
Creating table test_table.org_dir.foo.sun.com. ...
Adding entries to table ...
Group admins.foo.sun.com. membership is:
        myhost.sun.com.
Contents of Directory foo.sun.com. are:
        groups_dir
        org_dir

Contents of Table test_table.org_dir.foo.sun.com. are:
        Id.                     Name
        ---                     ----
        1                       John
        2                       Mary

Entry corresponding to id 1 is:
        1                       John

Starting to Clean up ...

Removing principal myhost.sun.com. from group admins.foo.sun.com.
Removing admins.foo.sun.com. group from namespace ...
Deleting all entries from table test_table.org_dir.foo.sun.com. ...
Deleting table test_table.org_dir.foo.sun.com. itself ...
Removing groups_dir.foo.sun.com. directory object from namespace ...
Removing org_dir.foo.sun.com. directory object from namespace ...
Removing foo.sun.com. directory object from namespace ...
myhost%
```

Figure 5-3 NIS+ Program Execution

As a debugging aid, the same operations are performed by the following command sequence. The first command:

```
% niscat -o `domainname`
```

displays the name of the master server. Substitute the master server name where the variable *master* appears below.

```
% nismkdir -m master foo.`domainname`.

# Create the org_dir.foo subdirectory with the specified master
% nismkdir -m master org_dir.foo.`domainname`.
# Create the groups_dir.foo subdirectory with the specified master
% nismkdir -m master groups_dir.foo.`domainname`.
# Create the "admins" group
% nisgrpadm -c admins.foo.`domainname`.

# Add yourself as a member of this group
% nisgrpadm -a admins.foo.`domainname`. `nisdefaults -p`

# Create a test_table with two columns : Id and Name
% nistbladm -c test_data id=SI Name=SI \
test_table.org_dir.foo.`domainname`

# Add one entry to that table.
% nistbladm -a id=1 Name=John test_table.org_dir.foo.`domainname`.
# Add another entry to that table.
% nistbladm -a id=2 Name=Mary test_table.org_dir.foo.`domainname`.

# List the members of the group admins
% nisgrpadm -l admins.foo.`domainname`.
# List the contents of the foo directory
% nisls foo.`domainname`.
# List the contents of the test_table along with its header
% niscat -h test_table.org_dir.foo.`domainname`.

# Get the entry from the test_table where id = 1
% nismatch id=1 test_table.org_dir.foo.`domainname`.

# Delete all we created.
# First, delete yourself from the admins group
% nisgrpadm -r admins.foo.`domainname`. `nisdefaults -p`
# Delete the admins group
% nisgrpadm -d admins.foo.`domainname`.
# Delete all the entries from the test_table
% nistbladm -r "[],test_table.org_dir.foo.`domainname`."
# Delete the test_table itself.
```

```
% nistbladm -d test_table.org_dir.foo.`domainname`.
# Delete all three directories that we created
% nisrmdir groups_dir.foo.`domainname`.
% nisrmdir org_dir.foo.`domainname`.
% nisrmdir foo.`domainname`.
```

5