The CCSO Nameserver – Programmer's Guide

by
Steven Dorner   s–dorner@uiuc.edu
Computer and Communications Services Office
University of Illinois at Urbana

December 22, 1988

updated by
Paul Pomes   paul–pomes@uiuc.edu
Computer and Communications Services Office
University of Illinois at Urbana

August 2, 1992

Introduction

It is our intention that other institutions be easily able to use the CCSO Nameserver if they wish to do so. This document should provide most of the information necessary to use and modify the Nameserver for use at places other than the University of Illinois.

It is assumed that the reader is familiar with the material presented in *The CCSO Nameserver, A Description*, and *The CCSO Nameserver, Guide to Installation*. Those documents describe in some detail what the CCSO Nameserver is, and of what it consists. Readers familiar with the CSNet Name Server will also want to read *The CCSO Nameserver, Why* to see the major differences between CSNet's server and our own. This document will attempt to supplement the information in the abovementioned papers, chiefly in the areas of data structures and file formats, although other topics will be mentioned briefly.

Acknowledgment

The CCSO Nameserver is similar to the CSNet Name Server. This similarity is not accidental; the CCSO Nameserver is derived from the CSNet program, and still uses a good portion of the CSNet source code. We are grateful that the CSNet Name Server was made available to us.

Data Structures

Herein described is every structure used by the Nameserver, what it looks like, where it is defined, and where it is used. From these descriptions, you will infer that the Nameserver assumes that a `short` is two bytes, an `int` is four bytes, a `long` is four bytes, and a pointer is four bytes. If you intend to run the Nameserver on a machine that is set up differently, you would do well to take a good look at each data structure, especially those that deal with the database entries and indices themselves. While an effort has been made to make the code automatically adjust to differing word sizes, it has never been tried on an 8086, a Harris, or a Cray, so you are on your own. You should be especially careful to ensure that where the Nameserver uses a `long`, you give it at least four bytes with which to work.

Converted to portable n/troff format using the -me macros from funky Next WriteNow format (icch).

That said, on to the descriptions. Each description includes the declaration of the structure (lifted from the Nameserver source code).

ARG — Command Argument — include/commands.h

```
struct argument
{
    int     aType;
    int     aKey;
    char    *aFirst;
    char    *aSecond;
    FDESC   *aFD;
    struct argument *aNext;
    int     aRating;
};
typedef struct argument ARG;
```

Used in

```
      qi/add.c        qi/change.c     qi/query.c
      qi/auth.c       qi/commands.c   qi/set.c
```

The `ARG` structure is used by the Nameserver central server, *qi*, to hold the arguments to Nameserver commands. Each command is broken into words, and these words put into `ARG` structures for manipulation by the server.

The `aType` field is used to label each argument. This field is formed by or'ing together the appropriate bits (bits defined in `include/commands.h`). Meaningful combinations of bits are:

Bits	Example	**aFirst**	**aSecond**	Explanation
COMMAND	query	"query"	NULL	The name of a command.
RETURN	return	"return"	NULL	A return or make token.
VALUE	smith	"smith"	NULL	A field value or field name.
VALUE | EQUAL	email=	"email"	NULL	Make a field empty.
VALUE | EQUAL | VALUE2	name=john	"name"	"john"	A field and a value.

The actual command, token, or values of the arguments are pointed to by `aFirst` (`COMMAND, RETURN, VALUE`) and by `aSecond` (`VALUE2`). They point to “malloc-space”,[1] and are freed at the end of each command.

The next argument in the command line is pointed to by `aNext`, unless we are at the end of the command, in which case `aNext` is `NULL`.

If an argument refers to a field name (such as a field on which to query, or a field to be printed by a query), `aFD` will point to the `FDESC` for the field with the name `aFirst` (if there is no field with the given name, the command will be discarded.)

`AKey` and `aRating` are used when the argument is a field and value to be looked for during a query. `AKey` will be set to 1 if the field in question is an indexed field. `ARating` is computed for indexed fields, and is a measure of how easy it would be to find entries based on the argument. The primary criterion here is lack of metacharacters; length of the value to be looked for is given second priority.

[1] Storage dynamically allocated via the UNIX library function *malloc*(3).

CMD — Command Handling Information — ph/ph.c

```
struct command
{
    char    *cName;          /*the name of the command */
    int     cLog;            /*must be logged in to use? */
    int     (*cFunc) ();     /*function to call for command */
};
typedef struct command CMD;
```

Used in `ph.c`.

The Nameserver client, *ph*, knows its commands from a table. The table is made up of `CMD` structures. The elements are pretty straightforward; the name of the command (`cName`), a flag indicating whether or not the user must be logged in to use the command (`cLog`), the function that handles the command (this function should take two arguments; a pointer to the line the user typed and a flag indicating whether the command should be executed (0) or detailed help should be printed (1)) (`cFunc`).

QDIR — Values From A Nameserver Entry — include/qi.h

```
typedef char **QDIR;
```

Used in

```
qi/add.c      qi/commands.c qi/lookup.c   util/makei.c
qi/auth.c     qi/dbm.c      qi/query.c    util/mdump.c
qi/change.c   qi/field.c    util/maked.c
```

Probably the most basic structure of all is the `QDIR`. It is a pointer to an array of pointers, each pointer pointing to a field from a Nameserver entry. The pointer array is terminated with a `NULL` pointer. The fields each begin with the ASCII value of the `fdId` field of the `FDESC` that describes their data, followed by a colon, followed by the field's data, and terminated with a `NULL` byte. The pointer array may come from any of the suitable storage classes; the storage for the fields is almost always in malloc-space.

directory_entry — Information On the Current Entry — qi/dbm.c

```
struct directory_entry
{
    long    ent_index;
    DREC    *ent_ptr;
};
```

Used in `qi/dbm.c`.

The database portion of the Nameserver central server operates on the “current entry”, with commands to make a given entry current, and to do various things to that entry. The number (in the *.dir* file) of the entry so selected (`ent_index`), and a pointer to the data from that entry (`ent_ptr`, which points to a `DREC`), is kept in a directory_entry structure in `qi/dbm.c`. The structure is not used elsewhere.

dirhead — Header of the .dir File — include/db.h

```
struct dirhead
{                            /* in block 0 of the .dir file */
    PTRTYPE nents;           /* number of entries in the .dir file */
    PTRTYPE next_id;         /* the next id capable of being issued */
    int     hashes[NHASH];   /* # of hashes to find index entries */
    int     nfree;           /* number of free entries in freelist,
                              * (not currently used) */
    int     freel[10];
};
```

Used in

```
      qi/dbi.c      util/border.c util/makei.c
      qi/dbm.c      util/credb.c  util/mdump.c
```

and in the *.dir* and *.dov* files.

The *.dir* file contains the data for Nameserver entries. The first part of that file is the header, and it is read and written directly to and from a `dirhead` structure. Thus, this structure is incarnate both in memory and on disk. (On disk, it is padded at the end to the size of a DREC, 256 bytes.)

Undoubtedly the most often used part of this structure is `nents`, which gives the total number of Nameserver database entries. It is especially popular with Nameserver utilities, who like to know how many entries they must process. Both `nents` and `next_id` are used when new Nameserver entries are added to the database. The free count (`nfree`) and the free list (`freel`) are not currently being used. The `hashes` array is a histogram of the number of indexed strings requiring a given number of applications of the hashing function. This has little to do with *.dir* file, but is kept here for convenience.

dumptype — Database Dump Names & Functions — util/mdump.c

```
struct dumptype
{
    char    *name;
    int     (*select) ();
    int     (*dump) ();
};
```

Used in `mdump.c`.

Mdump is a program to dump the contents of the Nameserver database into an ASCII file. Many different dumps are provided; they differ in which entries are dumped, and what fields are dumped from each entry. *Mdump* uses an array of `dumptype` structures to keep track of the different dumps. Each dump has a name (`name`), a function that is called to determine whether or not to include a given entry in the dump (`select`, called with a QDIR pointer for the entry), and the action to take for selected entries (`dump`, called with a QDIR pointer for the entry). This design permits *mdump* to be very modular, and has made customized dumping of the database a trivial task.

DOVR — Overflow of Entry Data — include/db.h

```
struct d_ovrflo
{
    char    d_mdata[NDOCHARS];
    PTRTYPE d_nextptr;          /* ptr to next ovrflo block */
};
typedef struct d_ovrflo DOVR;
```

Used in `qi/dbd.c`, and in the *.dov* file.

The *.dir* file is made up of fixed length records (DREC). Entries that are too long to fit in a DREC are continued in one or more DOVR records. The DOVR structure is read and written directly to the *.dov* file, and hence is used both in memory and on disk. The format is very simple; all but the last word are used for entry data (`d_mdata`). The last word (`d_nextptr`) is either the number of the next DOVR used by this entry, or NULL if the entry is completed in this block.

DOVR structures are used only when reading or writing entries; most entry manipulation takes place in QDIR or DREC structures.

DREC — Entry Data — include/db.h

```
struct d_record
{
    PTRTYPE d_ovrptr;           /* ptr to ovrflo block ( if any ) */
    PTRTYPE d_id;               /* unique id */
    long    d_crdate;           /* date of creation */
    long    d_chdate;           /* date of last modification */
    unsigned short d_dead;      /* deleted entry */
    unsigned short d_datalen;   /* length of data that follows */
    char    d_data[NDCHARS];    /* various strings, variable length */
};
typedef struct d_record DREC;
```

Used in

```
       qi/dbd.c       qi/dbm.c     util/credb.c,
```

and in the *.dir* file.

Each Nameserver entry (on disk) begins with a DREC. If all the data in the entry cannot be contained in one DREC (on disk), DOVR structures will be used to contain the remaining data. The DREC is used somewhat differently in memory. When an entry is read in, the DREC is first read from the *.dir* file; if there are overflow blocks, the DREC is **lengthened** to accommodate the excess data. Therefore, while a DREC is 256 bytes on disk, in memory it may be much larger.

`D_ovrptr` is the number of the first overflow block (DOVR) for this entry, or NULL if there are no overflow blocks. `D_id` is the number of the DREC in the *.dir* file. `D_crdate` is the creation date of the entry, and `d_chdate` is the date the entry was last changed; both dates are in seconds since the UNIX epoch (00:00 GMT Jan 1, 1970). If `d_dead` is non-zero, the entry should be ignored. `D_datalen` is the number of bytes of data in the entry; this includes space for NULL terminators for fields, but not space for any of the header fields or pointers; it is the length of the data alone. Finally, `d_data` is the entry's data; on disk, the data may be continued in DOVR structures; in memory, the DREC will be lengthened as mentioned above.

Within a DREC, the data is organized into fields. Each field is a null-terminated ASCII string, prefixed by a tag consisting of the `fdId` of the FDESC for the field (in ASCII) and a colon. There may be an essentially unlimited number of fields in a single entry. Only one field tagged with any given FDESC should appear in an entry.

FDESC — Field Description — include/field.h

```
struct fielddesc
{
    short fdId;        /* id # of the field */
    short fdMax;       /* maximum length of the field */
    int   dIndexed;    /* do we index this field? */
    int   fdLookup;    /* do we let just anyone do lookups with this? */
    int   fdPublic;    /* is field publicly viewable? */
    int   fdDefault;  /* print the field by default? */
    int   fdAlways;   /* print the always fields ? */
    int   fdAny;      /* the search field/property any */
    int   fdTurn;     /* can the user turn off display of this field? */
    int   fdChange;   /* is field changeable by the user? */
    int   fdSacred;   /* field requires great holiness of changer */
    int   fdEncrypt;  /* field requires encryption when it passes the net */
    int   fdNoPeople; /* field may not be changed for "people"
                       * entries, but can for others */
    int   fdForcePub; /* field is public, no matter what F_SUPPRESS is */
    char  *fdName;    /* name of the field */
    char  *fdHelp;    /* help for this field */
    char  *fdMerge;   /* merge instructions for this field */
};
typedef struct fielddesc FDESC;
```

Used in

```
      include/field.h qi/change.c     qi/field.c      qi/query.c
      qi/auth.c       qi/commands.c   qi/lookup.c
```

Each Nameserver entry is made up of one or more *fields*. Each field has associated with it a `FDESC` that describes the data in the field. A `FDESC` consists of a unique number that identifies the field (`fdId`), a maximum length for the field (`fdMax`), a name for the field (`fdName`), some description of what the field is intended to contain (`fdHelp`), instructions on how the field is to be merged during updates (`fdMerge`), and a set of attributes for the field. The attributes and their meanings are as follows:

`fdIndexed`	Words from this field appear in the Nameserver index (hash table in the *.idx* file). Any command that selects Nameserver entries must specify at least one field that is indexed as part of its search criteria.
`fdLookup`	This field may be specified in a lookup. That is, it is permissible to use the contents of this field as a method for selecting entries. Most fields have this attribute; it is present for the rare case where it may be desirable to turn it off.
`fdPublic`	Fields with this attribute may be viewed by anyone. Some fields (like the password field, for example) are private to the owner of the entry in which they appear, and should not be shown to the general public. Such fields would have the `fdPublic` attribute turned off.
`fdDefault`	With this attribute turned on, the field will be printed when a query is issued that does not specify which fields are to be returned.
`fdAlways`	When enabled, this attribute forces the field's contents to be always printed in addition to whatever fields specified by the query.
`fdAny`	This field is always searched by queries.
`fdTurn`	The field may be inhibited from display to the public by putting an asterisk as the first character of the field. This is not currently implemented usefully.
`fdChange`	The field's contents may be changed by anyone who knows the password for the entry in question.
`fdSacred`	This attribute is not in current use, but exists for historical reasons.
`fdEncrypt`	The contents of this field should be encrypted before being transmitted over a network.
`fdNoPeople`	The contents of the field may not be changed for entries that have a type of "people" but can be for other types.
`fdForcePub`	Force the contents of the field to be Public no matter what `F_SUPPRESS`'s value is (field "suppress" in the *cnf* file).

QHEADER — Header of .idx File — include/bintree.h

```
struct header
{
    IDX   seq_set;      /* pointer to first leaf */
    IDX   freelist;     /* unused */
    IDX   last_leaf;    /* pointer to last leaf */
    IDX   index_root;   /* pointer to first node */
    int   reads;        /* statistics... */
    int   writes;       /* statistics... */
    int   lookups;      /* statistics... */
    int   inserts;      /* statistics... */
    int   deletes;      /* statistics... */
};
typedef struct header QHEADER;
```

Used in

```
qi/bintree.c        util/build.c       util/border.c
util/maket.c
```

and in the *.seq* file.

A `QHEADER` is found as the first part of the *.seq* file. This file contains a linked list that holds all the strings in the Nameserver index (*.idx* file) in lexicographic order. `Seq_set` is the number of the first chunk of the linked list (these "chunks" are actually `LEAF` structures, and may contain one or more `ITEM`'s, which in turn contain the index strings and the index number for the strings). `Freelist` is the number of the first unused `LEAF` in a string of unused `LEAF`'s. The element `index_root` actually refers to the *.bdx* file, and is the number of the top of the tree of `NODE`'s contained in that file. What follows are statistics; they are not currently being used.

iindex — Hash Table Index Entry — include/db.h

```
struct iindex
{
    union
    {
        char    ii_string[NICHARS];
        PTRTYPE ii_recptrs[NIPTRS];
    } i_i;
};
```

Used in

```
qi/dbi.c      util/build.c      util/credb.c
```

and in the *.idx* file.

The `iindex` structure is the basic component of the Nameserver's hash table index. An `iindex` structure is really both variants (`ii_string` and `ii_recptrs`) at the same time. From the beginning of the structure to the first `NULL` byte, it is a string from the Nameserver database. From the first full word after the word in which the `NULL` byte appears, it is a list of entry numbers where the word appears, until the first `NULL` word or the last word in the structure. The last word in the structure, if not `NULL`, is the number of the overflow block that continues this index entry.

LEAF — Element of List of Hash Table Strings — include/bintree.h

```
struct leaf
{
    IDX     leaf_no;        /* this leaf's index */
    IDX     next;           /* pointer to next leaf */
    int     n_bytes;        /* number of bytes in data */
    char    data[DATA_SIZE]; /* data--zero or more ITEMs */
};
typedef struct leaf LEAF;
```

Used in

```
      qi/bintree.c      util/border.c     util/maket.c
```

and in the *.seq* file.

The `LEAF` is used to maintain a linked list of all the strings in the Nameserver index (*.idx* file), in lexicographic order. This list is useful for searching the index itself (as opposed to using the index to search the database). Each `LEAF` has a number (`leaf_no`), the number of the next `LEAF` in the list (`next`), some data (`data`), and the length of the data (`n_bytes`).

The data consists of one or more `ITEM`'s; each `ITEM` contains the number of the index entry involved, and the string in that entry. `ITEM`'s are stored in order within a `LEAF`; thus, all the strings in the Nameserver index may be examined in order by looking at each `LEAF` in order, looking at each `ITEM` of each `LEAF` in order. `ITEM`'s end with a `NULL` index entry number; there is no fixed number of `ITEM`'s in a `LEAF`.

LEAF_DES — Information About a LEAF — include/bintree.h

```
struct leaf_des
{
    IDX     leaf_no;            /* start of leaf string */
    char    max_key[KEY_SIZE];  /* biggest key in leaf string */
};
typedef struct leaf_des LEAF_DES;
```

Used in `util/build.c` and `util/maket.c`.

The `LEAF_DES` structure is only used while building the *.bdx* file. Its sole function is to keep track of the lexicographically greatest string in each leaf. `Max_key` holds the first four letters of the greatest string, and `leaf_no` is the number of the leaf in question.

NODE — Nodes of Tree Built From LEAF's — include/bintree.h

```
struct node
{
    IDX     l_ptr;          /* if your name is <= key */
    char    key[KEY_SIZE];  /* greatest key in l_ptr subtree */
    IDX     r_ptr;          /* if your name is > key */
};
typedef struct node NODE;
```

Used in

```
      qi/bintree.c  util/border.c util/build.c  util/maket.c
```

and in the *.bdx* file.

Searching the linked list of `LEAF`'s can be quite time-consuming; the *.bdx* file, made up of `NODE`'s, is used to quickly find the proper starting point for searches. Each `NODE` contains the first four letters of an index string (key), the number of the NODE or LEAF containing strings less than or equal to the key (l_ptr), and the number of the NODE or LEAF containing strings greater than or equal to the key (r_ptr). In this context, a negative number means a LEAF is being pointed to, and a positive number means another NODE is being pointed to.

OPTION — The Name And Value of a Nameserver Option — include/options.h

```
struct option
{
    char    *opName;
    char    *opValue;
};
typedef struct option OPTION;
```

Used in qi/qi.c and qi/set.c.

This one is pretty simple. Nameserver options are kept in an array of OPTION structures. Each structure has the name of the option (opName, in static data), and the value of the option, or NULL if the option is not set, (opValue, in malloc-space).

suffix — File Suffix and Selector Mask — util/border.c

```
struct suffix
{
    char    *suffix;
    int     mask;
};
```

Used in util/border.c.

This structure is used to keep track of the six suffices (*dir*, *dov*, *idx*, *iov*, *seq*, and *bdx*) that are used for Nameserver files. The suffix string is kept in suffix, and a bit that is used for selecting a particular suffix is kept in mask; a bit pattern is generated from *border*'s arguments, and mask is anded with that pattern to see if the file with the particular suffix is to be reordered.

File Organization

The Nameserver database is kept in six files. The files and their functions are:

.dir The first part of every entry is kept in the *.dir* file. The file begins with a dirhead and has one DREC for every Nameserver entry.

.dov Those entries too big to fit into a single DREC are continued in the *.dov* file. Its entries are of type DOVR; like the *.dir* file, it begins with a dirhead.

.idx The Nameserver's hash table is kept here. It begins with a QHEADER, and continues with iindex structures.

.iov Index entries too long for one iindex are continued in the *.iov* file (an index entry becomes too long if the string it references appears in many Nameserver entries; “smith”, for example, has multiple continuations). Each entry is a list of pointers, all but the last being pointers into the *.dir* file; the last pointer is a pointer to further index overflow blocks. If the block is not filled, the last valid pointer will be followed by a NULL pointer. The zeroth entry in the *.iov* file is empty.

.seq This file contains every string in the Nameserver index, in lexicographic order. It is used during metacharacter searches, and consists of LEAF structures, each containing one or more ITEM's. The first leaf in the linked list is pointed to by the seq_set element of the QHEADER, found in the *.idx* file.

.bdx The *.bdx* file contains a tree that speeds the searching of the *.seq* file. This tree is made up of NODE structures; the top of the tree is pointed to by the index_root element of the QHEADER, found in

the *.idx* file.

To better understand the organization of Nameserver files, consider a database consisting of only the following data (the → symbol represents the tab character):

```
3:Anna Arcola Anderson→0:142 Aspen Avenue Arcadia
  Alaska→10:All-Around Architect and
  Annunciator→9:Archeology Anthropology and Alimentary
  Angles→15:Asking All American Armenians About Asps
  Alligators Antelopes and Alphonse Amato→16:Avid Activist
  for All-merican Amateur Arrest Association

3:Crispin C Caramel→0:52C Calle Cadiz Cropcount
  California→10:Creepy-Crawly-Creature Creator

3:Dexter D Dripslobber→0:224 Deerdropping Drive Denver
  Delaware→10:Decimator of Delinquent Drivers
```

Once we have turned this data into a Nameserver database, named "example", let's look up the string "142", and see how the Nameserver would go about locating it.

The following diagram shows the relevant portions of the example database. Important addresses and values are show in solid boxes; interesting but incidental information is shown in dashed boxes. The "#" symbol represents a `NULL` byte.

```
   -----------------------------<i<-------------------------
   |                                                       |
  \|/                                                      |
0x00000  #  #  #  #  #  #  #  #  #  #  #  # ff ff ff ff    |  example.bdx
0x00010  c  r  e  f ff ff ff ff   . . .     ^^^^^|^^^^^    |
         ^^^^^^^^^^------------->ii>-------------|         |
                                                \|/       /i\
                                                 |         |
   -----------------------------<iii<-------------         |
   |                                                       |
  \|/                                                      |
0x00100  #  #  # 01  #  #  # 02  #  #  # E6  #  #  # 0C    |  example.seq
                                             ^^^^^^^^^^--  |
0x00110  1  4  2  #  #  #  # 1D  2  2  4  #  #  #  # 19 |  |
                                                        |  |
                                                        |  |
   -----------------------------<iv<---------------------  |
   |                                                       |
   |                                                       |
0x00000  #  #  #  #  #  #  #  #  #  #  #  #  #  #  # 00    |  example.idx
  \|/    . . .                               ^^^^^^^^^^-----
0x00300  1  4  2  #  #  #  # 01  #  #  #  #  #  #  #  #
                     ^^^^|^^^^^
                         |
   ---------<1<-----------
   |
  \|/
0x00100  #  #  # 01  #  #  # 01  # A8 17 B8  # A8 17 B8       example.dir
         ^^^^^^^^^^------------------->2>-----------------
0x00110  #  # 01 22  3  :  A  n  n  a     A  r  c  o  l  |
0x00120  a     A  n  d  e  r  s  o  n     0  :  1  4  2  |
0x00130     A  s  p  e  n     A  v  e  b  u  e     A  r  |
0x00140  c  a  d  i  a     A  l  a  s  k  a  #  1  0  :  |
0x00150  A  l  l  -  A  r  o  u  n  d     A  r  c  h  i  |
0x00160  t  e  c  t     a  n  d     A  n  n  u  n  i  c  |
0x00170  a  t  o  r  #  9  :  A  r  c  h  e  o  l  o  g  |
0x00180  y     A  n  t  h  r  o  p  o  l  o  g  y     a  |
0x00190  n  d     A  l  i  m  e  n  t  a  r  y     A  n  |
0x001a0  g  l  e  s  #  1  5  :  A  s  k  i  n  g     A  |
0x001b0  l  l     A  m  e  r  i  c  a  n     A  r  m  e  |
0x001c0  n  i  a  n  s     A  b  o  u  t     A  s  p  s  |
0x001d0     A  l  l  i  g  a  t  o  r  s     A  n  t  e  |
0x001e0  l  o  p  e  s     a  n  d     A  l  p  h  o  n  |
0x001f0  s  e     A  m  a  t  o  #  1  6  :  A  v  i  d  |
 . . .                                                   |
                                                         |
   -------------------------<2<---------------------------
   |
  \|/
0x00100     A  c  t  i  v  i  s  t     f  o  r     A  l       example.dov
0x00110  l  -  A  m  e  r  i  c  a  n     A  m  a  t  e
0x00120  u  r     A  r  r  e  s  t     A  s  s  o  c  i
0x00130  a  t  i  o  n  #  #  #  #  #  #  #  #  #  #  #
```

1 Compute the hashing function for the string "142". The result points to location 0x300 in the *.idx* file. In that `iindex`, we find the string "142", indicating that this is indeed the `iindex` we want. The next full word is 1, indicating that the string "142" appears in the first entry in the *.dir* file. Notice that the word after our 1 is a full word of zero; this indicates that there are no more entries containing "142".

2 After following the pointer into the *.dir* file, we find the first database entry (DREC at location 0x100, after the `dirhead`). We notice from the first word in the entry (`d_ovrptr`) that the entry's data is continued in the first data block of the *.dov* file (at 0x100, after the `dirhead`). The next word (`d_id`) confirms that we are indeed at entry 1 in the *.dir* file, and the half word at 0x110 (`d_dead`) tells us by being NULL that the entry is in use. We notice that the data is 0x122 bytes long from the next half word (`d_datalen`). And sure enough, our string does appear in the entry, as part of the address field, between 0x12b and 0x14c.

Suppose that instead of looking for "142", we were looking for anything beginning with "14". Since we wouldn't know where our strings might hash, we must search the index to find strings that fit our pattern.

i First, we find the head NODE of the tree in the *.bdx* file. This is kept in the *.idx* file, in the `index_root` element of the QHEADER, and is the fourth word of the *.idx* file. In our case, this word is 0, indicating the tree begins with NODE 0.

ii NODE 0 in the *.bdx* file has as its key "cref" (at 0x10). Our goal string, "14", is less than "cref", so we follow the left pointer (`l_ptr`, at 0xc). It is −1, meaning the LEAF containing keys greater than or equal to our goal key is the LEAF 1.

iii The first LEAF (at 0x100) does indeed contain a string that matches "14"; the string is "142", and we notice (at 0x10c, which is the `p_number` of an ITEM) that the string "142" appears in the *.idx* file as number 0xc.

iv 0xc translates to an address of 0x300 in the *.idx* file; the process continues with steps 1 and 2 above.

Statistics and the Nameserver Log

The Nameserver logs every command and error that it sees via the 4.3BSD syslog facility. At our site, we "roll over" this log weekly, and keep information for one week back. A week's log file is typically half a megabyte or so (representing a few thousand Nameserver commands).

We use this log for several things. First, it tells us how much use our Nameserver gets; this allows us to judge user satisfaction. Second, it tells us where our Nameserver is used from; this lets us know if we are getting good penetration into the computing community, or if our service is unknown to some parts of the campus. It also allows us to detect possible abuses of the Nameserver; if a host suddenly makes thousands of queries, we can look at that host's commands to see if someone is trying to use the Nameserver as a mailing list, or overloading it with nonsense queries. Third, it tells us what commands users actually use, and what commands are gathering dust; that helps us allocate our time to areas of user interest, rather than spend our time improving something no one cares about anyway. Fourth, It tells us how users are doing with the Nameserver; if a high proportion of responses for a particular command are errors, it may mean we need to modify the command to make it more intuitive, or improve our documentation. Finally, it allows us to see exactly what a user has done when that user comes to us with a problem using the Nameserver. Usually, the log gives us the information we need to discover the user's problem.

The program that allows us to (in some measure) accomplish these wonders with the log file is in the subdirectory stats. The *nsstats* program is invoked by *cron*(8). Unlike much of the Nameserver, this program is quite informal, written to serve our needs only; the most apt word to use is "hack". But we have found it to be a useful hack, and perhaps you will, too.

nsstats

I'll present the output from *nsstats* in sections, each line preceded with a line number, and explain what the section means. Missing line numbers correspond to blank lines in the output.

```
ph stats Aug 10
```

The first line gives the day for which the statistics pertain (August 10th).

```
4480 sessions from 309 hosts.
```

The next line totals the number of Nameserver sessions (4480), and the number of different hosts from which the sessions originated (309).

```
uxa.cso.uiuc.edu          960 (21%)
vmd.cso.uiuc.edu          112 (2%)
uxc.cso.uiuc.edu          130 (2%)
garcon.cso.uiuc.edu       683 (15%)
ux1.cso.uiuc.edu          887 (19%)
other (304 hosts)         1708 (38%)
```

This section shows all hosts who had at least 50 Nameserver sessions that day, the number of sessions coming from each, and the percentage of the total number of sessions that number represents. Hosts making less than 50 queries are lumped together in the "other" category, with the number of such hosts placed in parentheses after the "other" label (in this case, there were 304 hosts who made less than 50 queries). This section is a good place to find potential Nameserver abuse; most hosts appearing here should be machines with a large user-base; single-person workstations making hundreds of queries is quite unusual.

```
308 commands used 18638 times
```

The next section lists the different commands and how many times they were used. First the total number of significant Nameserver commands (18638), as well as the number of **different** commands given (308). The latter number counts only command names, not arguments; "query john smith" and "query jane doe" are considered equivalent for this purpose.

```
ph              166 (0%)
email            49 (0%)
login:            8 (0%)
quit            738 (3%)
siteinfo          6 (0%)
status          118 (0%)
answer           58 (0%)
attempting       13 (0%)
login           146 (0%)
clear            39 (0%)
Password          7 (0%)
fields           33 (0%)
id             2532 (13%)
query          5141 (27%)
change          107 (0%)
accting          25 (0%)
help             80 (0%)
weather          13 (0%)
Done           4464 (23%)
begin          4480 (24%)
```

The individual commands are listed, followed by the number of times they were issued, and the percentage of commands that number represents. Note that some commands (such as "quit" and "id" are automatically generated once per Nameserver session); one must be somewhat cautious in interpreting the numbers here.

Everything You Always Wanted to Know, But Were Afraid to Ask

The next section answers some often-asked questions about the Nameserver. The information presented is admittedly fragmentary; it may be useful nonetheless.

How Do You Assign Passwords?

The Nameserver tries to be accommodating with respect to passwords. First, find the definition for Hero in `qi/commands.c`. If there is no entry with this string as an alias, anyone may use the add command to add entries to the database, including adding a Hero entry to the database. Once the Hero entry exists, normal security is in force.

Normal security means that, when a login is attempted to a given alias, the entry is fetched; if a password field exists in the entry, that value should be used as the Nameserver password. If no password field exists, the last 8 characters of the id field are used as the password. If no id field is present, the password for the entry is “secret”. The moral of the story is not to generate an entry with an alias field but no id or password.

Just What Is the Id Field Anyway?

At the University of Illinois, we use the id field as a unique, immutable tag for entries. When we receive updated information from our administrative branch, we need to know which entry in our database to which the information applies. A name is insufficient for this purpose; names not only change, but they can be ambiguous.

The University already has a unique number for each student, faculty member, or staff member; unfortunately, this number is most often the person’s social security number, and is considered fairly private information.

What Field Descriptions Can We Change?

The field descriptions in the supplied *prod.cnf* are broken into two categories; one that warns against changing the descriptions in it, and one that bears no such warning. The criteria for splitting the field descriptions is quite simple; if the number for the field description appears in field.h and is therefore used by number in the Nameserver source code, the field description is in the first, protected, category. Changes to such fields must be made with care, and only after looking at how they are used in the source. Changes to fields in the second category may be made with impunity, provided:

(1) you are willing to put up with inconsistencies you may thereby introduce (for example, shortening the maximum length of a field may leave entries in your database with values too long in those fields) and

(2) You don’t change the Indexed property. If you add or remove the Indexed property, you **must** rebuild the Nameserver database with makei and build.